UNIVERSITY OF
ILLINOIS LIBRARY
AT URBANA-CHAMPAIGN

The person charging this material is responsible for its return to the library from which it was withdrawn on or before the **Latest Date** stamped below.

Theft, mutilatian, and underlining af boaks are reasans for disciplinary actian and may result in dismissal from the University.

Ta renew call Telephone Center, 333-8400

UNIVERSITY OF ILLINOIS LIBRARY AT URBANA-CHAMPAIGN

MAR 22 1983

JAN 02 1987

DEC 15 1987

SEP 08 1989

L161—O-1096

SEP 1 2 1980

I 1 6 r
no. 1024
cop. 2

Math

UIUCDCS-R-80-1024

UILU-ENG 80 1719

THE METHODOLOGY OF KNOWLEDGE LAYERS FOR INDUCING DESCRIPTIONS OF SEQUENTIALLY ORDERED EVENTS

By

Thomas Glen Dietterich

May 1980

THE LIBRARY OF THE

AUG 13 1980

UNIVERSITY OF ILLINOIS
URBANA-CHAMPAIGN

DEPARTMENT OF COMPUTER SCIENCE
UNIVERSITY OF ILLINOIS AT URBANA-CHAMPAIGN · URBANA, ILLINOIS

UIUCDCS-R-80-1024

THE METHODOLOGY OF KNOWLEDGE LAYERS FOR INDUCING DESCRIPTIONS OF SEQUENTIALLY ORDERED EVENTS

BY

THOMAS GLEN DIETTERICH

A.B., Oberlin College, 1977
M.S., University of Illinois, 1979

THESIS

Submitted in partial fulfillment of the requirements for the degree of Master of Science in Computer Science in the Graduate College of the University of Illinois at Urbana-Champaign, 1979, and supported in part by the National Science Foundation, Grant No. NSF MCS 76-22940

Urbana, Illinois

Digitized by the Internet Archive
in 2013

http://archive.org/details/methodologyofkno1024diet

ACKNOWLEDGMENTS

I wish to thank my family and friends for their continuing support during my work on this thesis. Special thanks go to Bridgette Barry for listening to all of my frustrations and giving me the strength to continue with the project. I also wish to thank Margaret Cheney for providing valuable floor space and moral support during the final weeks of thesis production.

Thanks also go to my Eleusis-playing friends, especially to Jim Stern, whose secret rules have become examples in this thesis. I thank A.B. Baskin for his good questions and comments which led me to reconsider and improve my ideas concerning segmentation.

I gratefully acknowledge the financial support of the National Science Foundation under grant number MCS-76-22940.

My debts to Professor R. S. Michalski should be very evident in the pages that follow. He originally suggested the topic of sequential data analysis and its application to Eleusis. The basic idea for the decomposition algorithm was also his. I thank him very much for his suggestions and for taking special time out from his busy schedule to read and offer suggestions on this thesis.

TABLE OF CONTENTS

1. INTRODUCTION 1
   1.1 Background 1
   1.2 A Program for Eleusis 1
   1.3 Research Paradigm: Tool-building 2
   1.4 Structure of this Thesis 2

2. THE THEORY OF INDUCING DESCRIPTIONS OF SEQUENTIAL EVENT SETS 3
   2.1 Events and Sequences of Events 3
   2.2 The Description Language VL_1 4
   2.3 Descriptions and Predictions 5
   2.4 Description Models 7
   2.5 Descriptions Based on Segmentation 8
   2.6 Discovering Descriptions--VL_1 Induction Algorithms 8
      2.6.1 The A^q Algorithm 9
      2.6.2 The Decomposition Algorithm 11
      2.6.3 The Periodic Algorithm 15
   2.7 Relationship to Statistical Methods 16

3. THE METHODOLOGY: KNOWLEDGE LAYERS 18
   3.1 Description of the Methodology 18
   3.2 Applying the Methodology to Eleusis 19
      3.2.1 Description of Eleusis 19
      3.2.2 Design Steps for an Eleusis Tool 22
      3.2.3 Other Functions of the Eleusis Tool 26
   3.3 Comparison of the Knowledge Layered System With Other AI Systems 26
   3.4 Relationship to the Learning System Model 29

4. EVALUATION OF PROGRAM PERFORMANCE 32
   4.1 The Implementation 32
   4.2 Sample Runs 32
      4.2.1 Example 1 33
      4.2.2 Example 2 34
      4.2.3 Example 3 35
      4.2.4 Example 4 36
      4.2.5 Example 5 36
   4.3 Evaluation 37
   4.4 Areas of Further Research 38

5. CONCLUSION 39

REFERENCES 40

APPENDIX I 43

APPENDIX II 46

1. INTRODUCTION

1.1 Background

Work in the area of computer induction is characterized by a continuum from general, universally applicable methods to specific, problem-oriented methods. Very general induction techniques (e.g. Vere[36,37,38,39], Hayes-Roth[13,14,15,16], Hunt[18], and early work by Michalski[23,25,30]) use little or no domain knowledge and develop formally correct generalizations. Hayes-Roth and Vere, for example, develop descriptions which are maximally specific conjunctive generalizations of a single set of events. Michalski's early work involved developing a quasi-minimal multiple-valued logic expression in disjunctive normal form which discriminates one set of events from another. These generalizations, although formally interesting, are often not plausible generalizations for real-world problems. Methods near the middle of the general-to-specific spectrum include the work of Larson and Michalski on Inducel[20,21,27] and AQ11[19,28]. These general-purpose programs generalize using some domain-specific advice and user-supplied preference criteria. AQ11 has been applied to problems in plant pathology [8]. At the far end of the spectrum are highly specialized systems designed to solve particular problems. These systems, which use large amounts of domain-specific knowledge, have achieved high performance in the areas of learning fragmentation rules in mass spectroscopy [4,5,6], learning spectra in nuclear magnetic resonance [33], discovering mathematical concepts [22], and learning the rules of baseball [34,35].

An examination of this spectrum of methods leads one to two conclusions: first, that there is a direct relationship between problem-solving ability and the amount of knowledge supplied to the program, and second, that there is an inverse relationship between generality and problem-solving ability. The first conclusion is not surprising. In terms of knowledge theory [31], programs which have more knowledge are by definition able to produce more results faster and with greater precision than programs which have less knowledge. No amount of clever programming can be expected to overcome this fact.

The second conclusion, that expert performance and broad scope of application cannot co-exist, is more suspect. It is certainly true for existing systems. But when these special-purpose expert systems have been developed, the primary emphasis has been placed on getting the job done (and done correctly) rather than on developing a general, portable system. The algorithms, representations, and general "world model" have been designed with implicit knowledge of the potential application. But this does not imply that there are knowledge theoretic limits that would prevent the construction of a highly modifiable expert system. Certainly, a special-purpose system is required to compute a smaller class of results than a general-purpose system and can therefore be expected to succeed where a general-purpose system would encounter time or space limits. But it is possible that properly designed programs can be developed which permit special-purpose knowledge to be incorporated in a convenient and general way so that expert performance may be obtained without sacrificing generality of application.

1.2 A Program for Eleusis

This thesis describes an attempt to develop a program which provides problem-specific performance together with ease of modification for application to different problems. The program induces plausible descriptions for events which are ordered in a sequence. In particular, the program provides expert performance in the card game Eleusis[1,12]. Eleusis is an induction game in which players attempt to guess a secret rule invented by the dealer. The secret rule tells which cards are playable at any point in the game. The cards must be played in a linear sequence according to the secret rule, and the dealer gives no hints or information aside from indicating whether or not each play is correct. Thus, each Eleusis game provides a sequence of ordered events on which we can test our program's abilities. The program which is described in this thesis acts as an intelligent assistant to a human Eleusis player.

Generality is obtained by adhering to a knowledge discipline—the program is constructed as a layered learning system in which the top-most layers use problem-specific knowledge and the

bottom-most layers use only general induction knowledge (See Figure 1). To apply the program to closely related problems, the top two Eleusis-oriented layers may be removed and replaced by new layers which perform functions peculiar to the new problem. To apply the program to vastly different problems (which do not involve sequentially ordered data), all but the bottom-most layer may need to be rewritten. Expert-level performance is achieved by permitting the upper layers to make extensive use of domain-specific knowledge in whatever form is convenient.

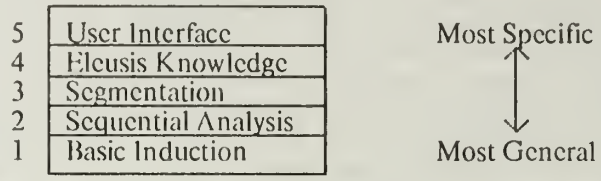


Figure 1. Layered Structure of Eleusis Program.

1.3 Research Paradigm: Tool-building

This research work has been guided by the tool-building paradigm. The goal of tool-building research is to develop effective computational tools which can be used by people to perform complex inference tasks. A good computational tool is general, powerful, and easily used and understood by the people who must use and maintain it. Few AI programs are good computational tools.

Among the issues raised by the tool-building approach are:

- The *comprehensibility principle*[9]. First articulated by Michalski, this principle states that a computational tool must present a conceptual interface which is understandable by the users of the tool. Michie [32] has pointed out some of the dangers of ignoring this principle.

- The tradeoff of generality and problem-solving ability. This thesis directs itself to techniques for trading off generality and effectiveness in learning systems.

- Knowledge engineering. How is knowledge to be placed in a computer? What balance of declarative and procedural, explicit and implicit knowledge should be provided? How can this knowledge be acquired and improved?

Research within the tool-building paradigm does not address several interesting areas of research. In particular, the work described in this thesis does not attempt to model psychological reality nor does it seek to create autonomous intelligent entities (artificial intelligences).

1.4 Structure of this Thesis

This thesis discusses three main topics. First, the problem of describing a sequence of events is investigated. The possible types of descriptions are defined and basic techniques for discovering these descriptions are detailed. The second major topic is the methodology of knowledge layers. The detailed designed of the Eleusis program is presented and compared to previous work in AI. Lastly, examples of the operation of the Eleusis program are given to demonstrate its strengths and weaknesses.

2. THE THEORY OF INDUCING DESCRIPTIONS OF SEQUENTIAL EVENT SETS

This chapter presents the theoretical background and the basic algorithms used to develop the Eleusis tool. The discussion is couched in general terms and the reader may wish to refer to Section 3.2.1 for a detailed account of the game of Eleusis in order to make these ideas concrete.

Throughout this thesis, the notations C, D, H, and S, are used to indicate the suits *clubs, diamonds, hearts,* and *spades.* Also, the letters A, J, Q, and K, are used to denote the *Ace, Jack, Queen,* and *King.* Consequently, the three of spades is denoted by '3S' and the king of hearts by 'KH'.

2.1 Events and Sequences of Events

This research seeks to construct a tool which can find plausible descriptions of a sequence of events. Imagine, for example, that some process is occurring in time—a process which we do not understand. We wish to understand the process by describing it in a way which permits us to predict the future course of the process from its past history. We want this to be a plausible description—conceptually simple and in accord with our knowledge of the problem at hand. In order to develop such a description, we could take regularly spaced "snapshots" of the process. We could measure, at each snapshot, the state of the process in terms of a set of variables which we believe are relevant or which may improve our understanding.

These measurements form a sequence of *events* which merely *represent* the original process. Since events are symbolic entities, they are amenable to manipulation by a computer. Formally,

> Definition 1: An *event* is a symbolic description of a set of measurements taken of some process, situation, or occurrence.
>
> Definition 2: A *sequential event set (sequential e-set)* is a set of events which are arranged in a totally ordered sequence. *Time-series events* are events whose ordering is based on the order in which they occur in time.

There may be many different representations of events. An event may be as simple as a single number, as elaborate as a graph or predicate logic description. The specific representation chosen for this research is a vector of symbols known as a canonical VL_1 complex. A canonical VL_1 complex is equivalent to an ordered n-tuple of symbols. Each symbol describes some measurement taken of the original process. (A definition of VL_1 appears below).

There can also be many different types of sequences of events. For example, time-series events need not be equally spaced in time. Sometimes *negative events* are available which indicate incorrect extensions of the sequence of events. In some cases, errors may be present in the data. Errors can be of three types: errors of ordering, of measurement, and of membership in the sequence. Ordering errors manifest themselves as out-of-sequence events. Measurement errors involve events which do not accurately represent the actual processes being described. Membership in the sequence is a form of classification error in which events have been included in or excluded from the sequence incorrectly. For the purposes of this research, the events comprising the sequence are considered to be equally spaced and error-free. The algorithms presented in this thesis work best when negative events are available, but satisfactory performance can be obtained without negative events.

It is beyond the scope of this thesis to handle sequential event sets which contain noise. Although many researchers have been criticized for ignoring noise, it was felt that there were plenty of difficult problems to solve in sequential data analysis without introducing noisy events as an additional feature. Error handling can be incorporated to some extent within the knowledge layer programming methodology. For example, errors of measurement can often be detected by using a knowledge-based preprocessing layer to filter them out. This approach is taken to some extent in Meta-DENDRAL [4,5,6] and in BASEBALL [34,35]. Noisy data admit many more plausible

descriptions than error-free data. In order to develop plausible descriptions of noisy data, either more search or more problem knowledge is required. It is an open question as to how such problem knowledge can be kept carefully separated from the general-purpose knowledge of the induction program and yet still be used effectively to eliminate noise-based descriptions.

2.2 The Description Language VL_1

The techniques and notation of VL_1 are used heavily in this thesis. VL_1 (Variable-valued Logic 1 [26,29,30]) is an extension of the propositional calculus (zeroeth-order logic) which uses the concept of a selector as the basic building block for propositions.

> Definition 3: A *selector* consists of a variable, a set of values called a *reference*, and a relation defined between the variable and the set of values.

Syntactically, a selector is written as

[*variable relation reference*]

An example of a selector is:

[suit = clubs, diamonds]

The variable is suit. The reference is {clubs, diamonds}, and the relation is =. This selector indicates that the suit variable may take on either of the values clubs or diamonds.

[size > 10]

This selector indicates that size must take a value greater than 10.

In any particular VL_1 system, each variable is defined to have an explicit set of values called its *domain*. All values which appear in the reference of a selector must be taken from the domain. For example, the domain of suit is {clubs, diamonds, hearts, spades}.

Each variable in a VL_1 system is also given a domain type which specifies the permitted generalizations of the variable. For example, the *interval* domain type indicates that any reference can be plausibly generalized by closing the interval between the smallest and the largest elements of the reference. Thus, the selector

[size = 2,5]

may be generalized to

[size = 2,3,4,5]

if *size* has an interval domain. Domain types have the very important function of providing problem-specific knowledge to the inductive program. In addition to interval domains, the Eleusis program supports:

- Nominal domains. All elements are unrelated and no plausible generalizations exist.
- Cyclic interval domains. The elements in a cyclic domain are circularly ordered so that end-around intervals are permitted. (Example: Card values are sometimes considered to be circular so that J Q K A 2 is a straight).

Intervals, both cyclic and normal, are denoted in the reference by writing the endpoints of the interval separated by two dots. Thus, [value = 2,3,4,5] is written as [value = 2..5], and [value = J, Q, K, A, 2] is written as [value = J..2].

Both events and descriptions can be conveniently represented by conjunctions of selectors called *complexes.*

Definition 4: A *complex* is a conjunction of selectors. It is written by placing selectors directly adjacent to each other:

[suit = clubs, diamonds][value < 3]

(This conjunction describes the cards {AC, 2C, AD, 2D}).

A *canonical complex* is a complex in which all variables are present, and all selectors have the = relation and a single value in the reference. A canonical complex describes a single entity—not a set.

In the context of sequential data analysis, we use a subscripting notation to indicate the ordering of various events. The subscript zero on a variable indicates that that variable refers to the current event of interest. A subscript of one refers to the event immediately preceding; a subscript of two, to the event before that; and so on. For example, [color1 = red][value0>6] indicates that the color in the preceding event was red and the value in the current event is greater than 6.

We also introduce so-called *difference* and *sum* variables. The variable dvalue01 has a value equal to value0-value1 (i.e. the difference of the values of the current card and the previous card). And the variable svalue01 takes on a value equal to the sum: value0 + value1.

We noted above that a canonical VL_1 complex is equivalent to an n-tuple of symbols. To use an n-tuple representation, all of the variables in a VL_1 system must be placed in some order. Then the elements in the n-tuple provide the references for each variable in that order. Thus, if we order the card variables as value followed by suit, the pair (10, clubs) is equivalent to the canonical VL_1 complex [value = 10][suit = clubs].

2.3 Descriptions and Predictions

How can a sequential event set be described? We seek descriptions which permit us to predict the future behavior of the sequence from past events. A description predicts an event if the event can be described by the description.

Definition 5: A *prediction* concerning an event E, is a description, D, of the set of possibilities for E along with some specification of the likelihood of each possibility. We write

D >--> E

when a description predicts an event. Note that this is a *nondeterministic* prediction in the sense that no single event is predicted, but instead a *set* of events—one of which must occur—is predicted.

In traditional fields, a statistical prediction specifies the possible values of some variable along with a probability distribution function which indicates the probability of each possible value. In the present work, a prediction is a logical description which subsumes all possibilities for the event in question. For example, a prediction that the next card will be red is merely the description

[color0 = red] along with the understanding that this is a perfect description (probability 1). There are two fundamental types of descriptions for sequential event sets which allow us to predict the future course of the sequence: *lookback* descriptions and *periodic* descriptions. A lookback description is a function, F, of the most recent events, which predicts the next event. If

$$S = \langle E_1, E_2, E_3, ..., E_n \rangle$$

is a sequence of events, then F can be applied to the l most recent events prior to any E_i in order to predict E_i:

$$F(E_{i-l}, E_{i-(l-1)}, ..., E_{i-2}, E_{i-1}) \text{>--> } E_i$$

l is called to lookback parameter. It indicates how far into the past it is necessary to look back in order to predict the next event. In a simple Markov process, for example, a lookback parameter of 1 is all that is ever required. An example of a lookback description is the function

$$F(x) = x+3$$

which describes the sequence

$$\langle 1, 4, 7, 10, 13, 16, 19, 22 \rangle$$

by predicting the next value in the sequence as a function of the previous value: $F(E_i) \text{>--> } E_{i+1}$.

A *periodic description* is a periodic function which describes each event in the sequence as a function of the position of that event in the sequence. For example, the periodic description

$$P(x) = x \bmod 4$$

describes the sequence

$$\langle 1, 2, 3, 0, 1, 2, 3, 0, 1, 2, 3, 0 \rangle$$

since

$$P(i) \text{>--> } E_i.$$

Since P is a periodic function, the function has a period or length, T, after which it repeats. The *phase* of an event is its relative position within the period. All events in the same phase have the same prediction. The sequence:

$$\langle 2C, 4H, 7C, AH, 6C, JH \rangle$$

may be described by the periodic function P:

$$P(i) = [suit0 = club] \quad \text{if } i \bmod 2 = 1$$
$$P(i) = [suit0 = heart] \quad \text{if } i \bmod 2 = 0.$$

All of the clubs are in the first phase of the period, and all of the hearts are in the second phase. A convenient way to specify the periodic function P is simply to list the descriptions of each phase as an ordered n-tuple. We could rewrite the above function as

$$P: ([suit0 = club], [suit0 = heart])$$

where it is understood that "[suit0 = club]" describes the first phase, and "[suit0 = heart]," the second.

2.4 Description Models

Induction is the process of finding plausible and useful descriptions of events. One approach to induction is to identify models which specify the form of plausible descriptions. Induction then becomes the two step process of first fitting data to a model and second evaluating the fit to assess the plausibility and utility of the resulting description. Such techniques have long been used in traditional regression analysis where the model is usually some specific regression polynomial. Statistical tests for goodness-of-fit have been developed for such models.

> Definition 6: A *model* prescribes the specific functional or syntactic form for a description.

Examples of description models are the decision tree used by Hunt [18], and the disjunctive normal form used by Michalski [23,25,30]. In a numerical sequence, a model might specify that the description is to be a lookback description in which the prediction is a linear function of the value of the previous number in the sequence:

$$F(x) = ax + b.$$

In this model, the a and b parameters need to be determined from the data. Obviously, the models used by a program carry a good deal of implicit problem-specific knowledge. It is important that a general inductive tool permit modification and manipulation of the models chosen.

Three models have been identified for use in Eleusis:

a. Periodic conjunctive model. This model specifies that the description must be a periodic description in which each phase is described by a single VL_1 complex. Example:

   Period ([color0 = red], [color0 = black])

   describes an alternating sequence of red and black cards.

b. Lookback decomposition model. This model specifies that the description must be a lookback description in the form of a disjunctive set of if-then rules:

   [color1 = red] => [value0<5] V
   [color1 = black] => [value0> = 5].

   The left-hand sides, or condition parts, of the rules must only refer to events prior to the event to be predicted (subscripts 1, 2, etc.). The right-hand sides provide predictions for the next event in the sequence given that the condition part is true.

   The decomposition model requires that the left-hand sides be disjoint—that only one if-then rule be applicable at any time. Furthermore, it is desirable that the right-hand sides should also be disjoint. The algorithm described below does not require right-hand side disjointness, however.

c. Disjunctive Normal Form (DNF). This lookback model requires only that the description be a disjunction of VL_1 complexes. An example is:

   [dsuit01 = 0] V [dvalue01 = 0]

which indicates that either the suit of the current card must be the same as the suit of the previous card, or the value of the current card must be the same as the value of the previous card.

From a logic standpoint, any decomposition rule (and many periodic rules) can be written in disjunctive normal form. The periodic and decomposition models are useful not because of their theoretical expressiveness or power, but because they assist in locating plausible descriptions quickly. The space of all DNF descriptions is very large and difficult to search.

2.5 Descriptions Based on Segmentation

Very often sequences of events are best described in a hierarchical fashion as a sequence of subsequences. For example,

$$S = \langle 3, 4, 4, 5, 5, 5, 6, 6, 6, 6, 7, 7, 7, 7, 7 \rangle$$

is best described as a sequence of subsequences. Each subsequence is a string of identical digits. The length of each subsequence is one longer than its predecessor. The digit used in the subsequence is one larger than the digit used in the previous subsequence. In VL_1, this can be indicated by the two part description:

Segmentation condition:	string = [dvalue01 = 0] :	(A)
Sequence description:	[dvalue01 = + 1][dlength01 = + 1]	(B)

Statement (A) defines a subsequence to be a string of adjacent events satisfying the constraint that their values must remain constant (dvalue01 = 0). The sequence is *segmented* into strings of maximal length satisfying this *segmenting condition.* This yields, in this example, the derived sequence

$$S' = \langle (3,1), (4,2), (5,3), (6,4), (7,5) \rangle$$

In S' we have used the n-tuple representation for VL_1 events. The first value in each ordered pair is the digit used in the corresponding string of events in S. The second value specifies the length of the corresponding string in S. Each ordered pair forms a new event in the derived sequence S'.

Once the sequence has been segmented, a DNF description, statement (B), can be written. In (B) dvalue01 and dlength01 refer to the values and lengths of the events in sequence S'.

Any of the description models listed in section 2.4 can be applied to a sequence after it has been segmented. The discovery of such segmented descriptions requires both the discovery of the segmentation condition and the discovery of the description of the segmented sequence.

2.6 Discovering Descriptions—VL_1 Induction Algorithms

How can these descriptions be discovered? In this section we outline the basic algorithms used to discover descriptions in the Eleusis program. The general approach is to choose a segmentation condition, a value for the lookback parameter, and a model. Then one of the VL_1 induction algorithms described in this section is called to fit the data to a model and assess the quality of the fit. The VL_1 algorithms are provided with events which have been developed by *transforming* the original sequence. As an example, consider the sequence of cards, S, shown in Figure 2.

S = <2C, 10D, 3S, AD, JC, 6H, 6C>.

Figure 2. Example Sequence of Cards.

Assume, for the moment, that no segmentation condition is applicable and that we are considering a lookback parameter of 1. This sequence of events can then be transformed into the VL_1 events listed in Table 1. Notice, using Table 1, that [dcolor01 = 1] for all events (i.e. color always changes from one card to the next). The VL_1 induction algorithms seek to discover exactly this sort of description.

The variables listed in Table 1 are called *derived variables* because they are derived from the original sequence. The events are *derived events.* The events in Table 1 are unordered. The original ordering of the sequence has been made explicit through the difference variables (dvalue, dsuit, and dcolor). Color is derived using knowledge of the characteristics of the cards.

Table 1. Transformed VL_1 Events

Each row in this table corresponds to one derived event, listed as an ordered 9-tuple. The meanings of each column are given in the top row of the table.

value1	suit1	color1	value0	suit0	color0	dvalue01	dsuit01	dcolor01
2	C	black	10	D	red	8	+1	1
10	D	red	3	S	black	-7	+2	1
3	S	black	A	D	red	-2	+2	1
A	D	red	J	C	black	10	+3	1
J	C	black	6	H	red	-5	+2	1
6	H	red	6	C	black	0	+2	1

In generating Table 1, value was given an interval domain, suit a cyclic interval domain (with the suits ordered as clubs, diamonds, hearts, spades, clubs, ...), and color a nominal domain (red or black). The difference variables reflect these domain types. Dvalue01 takes on values from -12 to +12, but dsuit01 takes values 0, 1, 2, and 3. Differences for cyclic interval domains are computed as values modulo n, where n is the size of the domain. Thus, the difference between clubs and hearts is +2 ((0-2) modulo 4 = 2). Dcolor01 is an example of a difference on a nominal variable. Dcolor is 0 if color0=color1, and 1 otherwise.

Table 1 could be used to discover DNF and decomposition descriptions with a lookback of 1, but it would not be useful for discovering periodic descriptions or descriptions with other lookbacks. Different derived variables and different events are required for discovering descriptions which fit different description models.

2.6.1 The A^q Algorithm. Much work in induction has been conducted by Michalski and his collaborators. Most of this work is based on the A^q algorithm [23,25,30] which was originally developed in the context of switching theory. This algorithm accepts as input a set of positive events and a set of negative events. Each event is a canonical VL_1 complex. A^q considers each VL_1 variable to be a variable in a multiple-valued logic covering problem. By developing a *cover* of the positive events against the negative events, A^q produces a description which is satisfied by all of the positive events and by none of the negative events. (A description *covers* an event if the event satisfies the description).

The process of developing a cover involves partially computing the complement of the set of negative events and intelligently selecting complexes which cover positive events. The final cover may be a single complex or a disjunction of complexes. A^q seeks to develop a disjunction with the

fewest number of complexes possible, but the algorithm is only quasi-optimal. It is capable, under certain conditions, of giving an upper bound on the distance from optimality of the solution it produces.

The algorithm proceeds in depth-first fashion by the method of disjoint stars. A positive event, e_j, is chosen and a star is built about e_j. A *star* is an approximation to the set of all prime implicants which cover e_j (and are in the complement of the set of all negative events). In other words, a star is the set of all complexes which cover event e_j, do not cover any negative events, and are maximal under inclusion. The best complex in the star, *lq*, is chosen to form part of the solution. The *best* complex is determined by applying a lexicographic evaluation functional (see appendix II). All events covered by *lq* are removed from further consideration. The process of choosing a positive event, e_j, building a star about e_j, and selecting the best element of the star is repeated. However, the new e_j must not have been covered by any element of *any previous star.* In this manner, disjoint, well-separated stars are built, and *lq*'s are selected. The process repeats until all events have been covered by at least one star. Some clean-up operations are required in the case where some positive events were covered by some star, but by no *lq.*

The process of building a star about an event e_j is simple. The disjunction of all negative events is complemented and then multiplied out, one event at a time. After each event is multiplied out, the set of intermediate products (so-called partial stars) is trimmed according to a user-specified preference criterion, and only the MAXSTAR best elements are retained. The user-specified preference criterion contains instructions to discard any complexes which do not cover any of the positive events. This prevents the algorithm from generating useless generalizations. The final star has MAXSTAR elements in it.

Note that all of the steps mentioned (complementation, multiplication, etc.) are being performed on variables which can take on a *set* of values. This is a multiple-valued covering process.

The strengths of the algorithm include

- flexibility—The user can specify the preference criterion which determines which *lq* is chosen from each star and which partial stars are retained during the star-building process. The algorithm can be modified to develop strictly disjoint complexes in the solution.

- optimality—If no trimming is performed, the algorithm provides a measure of how many additional complexes appear in the solution above and beyond the minimum necessary for the optimal solution. When this measure is zero, the algorithm has produced an *optimal* description in terms of the number of complexes in the description.

- quasi-optimality—The algorithm performs well even when the stars must be severely trimmed.

Unfortunately, the Λ^q algorithm is not sufficient for discovering all of the description models required for describing sequential event sets. Λ^q was designed to develop descriptions in disjunctive normal form with the fewest number of complexes. This is only one of the description models discussed above. Neither periodic nor decomposition descriptions can be easily discovered using Λ^q (although equivalent descriptions are sometimes discovered as DNF descriptions). Λ^q tends to develop a "lopsided" cover. Since it proceeds in a depth-first manner, it selects the largest prime implicant first and smaller prime implicants later. Descriptions involving symmetry (decomposition descriptions in particular) tend to be overlooked. Thus, although the Λ^q algorithm is powerful and useful, it does have some limitations which render it insufficient for completely solving the discovery problems of sequential event sets.

Two other algorithms have been incorporated into the Eleusis program: the decomposition algorithm and the periodic algorithm.

2.6.2 The Decomposition Algorithm. This algorithm seeks to fit the data to a decomposition model. It accepts as input a set of positive events and a set of negative events. In addition, some variables are designated as "left-hand side" variables (i.e. variables which describe previous events). A decomposition model seeks to explain the events in terms of the values of "left-hand side" variables. A decomposition description for the events in Table 1 would be

[color1 = red] => [color0 = black] V
[color1 = black] => [color0 = red].

This description *decomposes on* color1. It breaks the description of the sequence into two if-then rules. The => can be interpreted as an implication. The decomposition algorithm takes advantage of the constraints that both the left-hand and right-hand parts of the if-then rules must be single VL_1 complexes and that the left-hand sides must be disjoint.

The decomposition algorithm starts by performing a trial decomposition on each possible left-hand side variable. A trial decomposition for a given variable is formed by creating a complex for each possible value of the given variable (This basic idea was suggested to me by R. S. Michalski). All events covered by the given value of the given variable are merged together to form a complex. (The references of corresponding selectors are unioned). For example, using the events of Table 1, trial decompositions could be performed on value1, suit1, and color1 to yield the complexes shown in Table 2. The general idea is to form trial decompositions, choose the best decomposition, and break the problem into sub-problems, one for each if-then rule in the selected decomposition. The algorithm can then be applied recursively until a consistent description has been developed.

Table 2. Trial Decompositions.

On value1:

[value1 = A] => [value0 = J][suit0 = C][color0 = B]
[dvalue01 = 10][dsuit01 = +3][dcolor01 = 1]
[value1 = 2] => [value0 = 10][suit0 = D][color0 = R]
[dvalue01 = 8][dsuit01 = 1][dcolor01 = 1]
[value1 = 3] => [value0 = A][suit0 = D][color0 = R]
[dvalue01 = -2][dsuit01 = +2][dcolor01 = 1]
[value1 = 6] => [value0 = 6][suit0 = C][color0 = B]
[dvalue01 = 0][dsuit01 = +2][dcolor01 = 1]
[value1 = 10] => [value0 = 3][suit0 = S][color0 = B]
[dvalue01 = -7][dsuit01 = +2][dcolor01 = 1]
[value1 = J] => [value0 = 6][suit0 = H][color0 = R]
[dvalue01 = -5][dsuit01 = +2][dcolor01 = 1]

On suit1:

[suit1 = C] => [value0 = 6,10][suit0 = D,H][color0 = R]
[dvalue01 = -5,8][dsuit01 = +1,2][dcolor01 = 1]
[suit1 = D] => [value0 = 3,J][suit0 = C,S][color0 = B]
[dvalue01 = -7, +10][dsuit01 = +2,3][dcolor01 = 1]
[suit1 = H] => [value0 = 6][suit0 = C][color0 = B]
[dvalue01 = 0][dsuit01 = +2][dcolor01 = 1]
[suit1 = S] => [value0 = A][suit0 = D][color0 = R]
[dvalue01 = -2][dsuit01 = +2][dcolor01 = 1]

On color1

[color1 = R] => [value0 = 3,6,J][suit0 = C,S][color0 = B]
[dvalue01 = -7,0,10][dsuit01 = +2,3][dcolor01 = 1]
[color1 = B] => [value0 = A,6,10][suit0 = H,D][color0 = R]
[dvalue01 = -2,-5,8][dsuit01 = 1,2][dcolor01 = 1]

Table 2 shows the raw trial decompositions. These are very poor descriptions desriptions since they are complex and not sufficiently general. They must be processed further before a decision can be made as to which decomposition is best and should be refined. Three processing steps are applied to the trial decompositions.

The first processing step involves interval (and cyclic interval) variables such as value1. These variables often have many values and trial decompositions based on them are very uninteresting and implausible. (An Eleusis rule with 13 separate cases would be impossible to discover!) An attempt is made to close intervals on the left-hand side of the trial decomposition. Imagine, for example, that some sequence is well-described by the decomposition:

[value1<8] => [color0 = red] V
[value1>=8] => [color0 = black]

A trial decomposition would involve up to 13 different complexes for value1. The first processing step attempts to detect that all if-then rules below [value1 = 8] should be combined into one if-then rule, and that all if-then rules above [value1 = 7] should be combined into another if-then rule. The algorithm operates by computing distances between adjacent if-then rules and looking for

sudden jumps in the distance measure. Where a jump occurs (a local maximum), the algorithm tries to split the domain into cases.

The distance computation is a weighted multiple-valued Hamming distance. The weights are determined by taking user-specified plausibilities for each variable and adjusting these weights according to the discriminating power of each variable (taken singly). For instance, if a right-hand side variable is irrelevant in some if-then rule (i.e. its reference contains all possible values so that it is a "don't care" selector), then its weight is reduced to zero. As an example, assume we have the following two complexes and that the adjusted wieghts for suit, value and color are 0.50, 0.75, and 0.00.

complex 1:	[suit = C,D]	[value = 4..10]	[color = *]
complex 2:	[suit = C,S]	[value = 8..K]	[color = Black]
distance:	0.67	0.70	0.50
adjusted weight:	0.50	0.75	0.00

The distance between two selectors with references R and S is $1.0 - |R \cap S|/|R \cup S|$. Then the total weighted Hamming distance for this example is: 0.86.

The distances between adjacent if-then rules are computed and local maxima are located. If there is one maximum, the interval is split there, and two if-then rules are created. If there are two maxima, three if-then rules are created. If there are more than two maxima, the smaller maxima are suppressed.

Similar techniques are used for cyclic interval domains.

Once the cases have been determined, each trial decomposition is next processed by applying the domain-specific rules of generalization to the selectors on the right-hand sides of the if-then rules. Intervals are closed for interval variables and cyclic interval variables. Special domain types are defined for difference variables (variables derived by subtracting two other variables). The rules of generalization for difference variables attempt to find intervals about the zero point of the domain. Thus, [dvalue01 = –3,1,2] would be generalized to [dvalue01 = –3.. + 3]. One-sided intervals away from zero are also created: [dvalue01 = 3,4,6] would be generalized to [dvalue01>0]. These generalizations are only performed if the reference contains more than one value. Corresponding to the trial decompositions of Table 2 we get the generalized trial decompositions of Table 3. The notation [variable = *] is used when a variable can take on any value from its domain (i.e. it is irrelevant).

Table 3. Generalized Trial Decompositions.

On value1

[value1 = A..4] => [value0 = A..J][suit0 = C,D][color0 = *]
[dvalue01<>0][dsuit01<>0][dcolor01 = 1]
[value1 = 5..K] => [value0 = 3..6][suit0 = C,S][color0 = *]
[dvalue01< = 0][dsuit01 = 2][dcolor01 = 1]

On suit1

[suit1 = C] => [value0 = 6..10][suit0 = D,H][color0 = R]
[dvalue01<>0][dsuit01 = 1,2][dcolor01 = 1]
[suit1 = D] => [value0 = 3..J][suit0 = C,S][color0 = B]
[dvalue01<>0][dsuit01 = 2,3][dcolor01 = 1]
[suit1 = H] => [value0 = 6][suit0 = C][color0 = B]
[dvalue01 = 0][dsuit01 = 2][dcolor01 = 1]
[suit1 = S] => [value0 = A][suit0 = D][color0 = R]
[dvalue01 = -2][dsuit01 = 2][dcolor01 = 1]

On color1

```
[color1 = R] => [value0 = 3..J][suit0 = C,S][color0 = B]
        [dvalue01 = *][dsuit01 = 2,3][dcolor01 = 1]
[color1 = B] => [value0 = A..10][suit0 = H,D][color0 = R]
        [dvalue01<>0][dsuit01 = 1,2][dcolor01 = 1]
```

The third processing step examines the different if-then rules and attempts to make the right-hand sides of the rules disjoint by removing selectors which have overlapping references. Table 4 shows the results of this step. TRUE indicates that *all* selectors have been removed from the right-hand side so that *any* card is valid.

Table 4. Trial Decompositions With Overlapping Selectors Removed.
(Irrelevant selectors are omitted)

On value1
```
[value1 = A..4]  =>  TRUE
[value1 = 5..K]  =>  TRUE
```

On suit1
```
[suit1 = C]  =>  TRUE
[suit1 = D]  =>  TRUE
[suit1 = H]  =>  TRUE
[suit1 = S]  =>  TRUE
```

On color1
```
[color1 = R]  => [suit0 = C,S][color0 = B]
[color1 = B]  => [suit0 = D,H][color0 = R]
```

At this point, the algorithm has identified the rule fairly well. Now the best decomposition can be selected. The selection process uses a set of cost functions which measure characteristics of each trial decomposition. The cost functions are:

1. Count the number of negative examples that are incorrectly covered by this decomposition.

2. Count the number of cases (if-then rules) in this decomposition.

3. Return the user-specified plausibility for the variable being decomposed on.

4. Count the number of null cases for this decomposition (e.g. [value1 = 4] is a null case in Table 2).

5. Count the number of "simple" selectors in this decomposition. A simple selector can be written with a single value or interval in the reference (e.g. [value01>4] is a simple selector). After applying the generalization rules (as in Table 3) all selectors except those with nominal variables are necessarily simple.

The cost functions are applied in an ordered fashion using the functional sort algorithm developed by Michalski [26]. The trial decomposition with the lowest cost (according to these cost functions) is selected. Using the default cost functional, the lowest cost decomposition in Table 4 is the decomposition on color1. The other decompositions are completely overgeneralized.

The algorithm does not always proceed as indicated above. The user can request that the best trial decomposition be selected after performing only the first post-processing step, or after the second

post-processing step has been completed. In fact, it is recommended that the best decomposition be selected after the second step.

Once the best trial decomposition has been selected, it is checked to see if it is consistent with the events (covers no negative events). If it is, the decomposition algorithm terminates. If it is not, the problem is decomposed into separate subproblems, one for each if-then rule in the selected decomposition. Then the algorithm is repeated to solve these subproblems. The subproblems are solved simultaneously, so that the same variable is chosen for further decomposition in all subproblems.

The strengths of the decomposition algorithm are

- speed—The algorithm locates good decompositions quickly.
- aptness—The algorithm locates descriptions which fit the decomposition model very well.

The weaknesses of the algorithm are

- inability to produce alternatives—this is a depth-first algorithm which returns only one description. Often it is desirable to have a learning algorithm which returns a set of possible descriptions.
- restricted model—the algorithm was designed for a specific model. The generality of this model has not yet been demonstrated.

2.6.3 The Periodic Algorithm. The periodic algorithm is really just a modified version of the decomposition algorithm designed for discovering descriptions which fit the periodic model. A parameter is provided to the algorithm which indicates the number of phases to expect in the description. Each phase is treated as if it were a different if-then case in a trial decomposition. First, the events in each phase are combined to form a single complex (by forming the union of references of corresponding selectors). For the sequence S in Figure 2, the results are shown below. Note that no difference variables or variables describing previous events are included in these derived events.

phase 1: [value0 = 10,A,6][suit0 = D,H][color0 = R]
phase 2: [value0 = 3,J,6][suit0 = C,S][color0 = B]

If these complexes are consistent with the negative examples, then the references are generalized according to the domain types of the variables:

phase 1: [value0 = A..10][suit0 = D,H][color0 = R]
phase 2: [value0 = 3..J][suit0 = C,S][color0 = B]

If these generalized complexes are still consistent, selectors with overlapping references (overlapping with selectors in other phases) are removed:

phase 1: [suit0 = D,H][color0 = R]
phase 2: [suit0 = C,S][color0 = B]

If these complexes are still consistent, they are returned as the final description.

Both the periodic and the decomposition algorithms go through these two post-processing steps until the description becomes inconsistent. When this occurs, the algorithm backs up and returns the version of the description before it was overgeneralized to become inconsistent. If the first post-processing step leads to inconsistency, the star generation process of the Λ^q algorithm is invoked to attempt to extend the description against negative examples.

2.7 Relationship to Statistical Methods

There are many direct parallels between the previous discussion of sequential data analysis and the traditional area of time-series analysis.

Time-series events occur in many systems: the economy, the factory, the environment. Techniques have been developed to predict the future course of the time-series and to determine the appropriate amount of feedback required to control the system. The same sorts of descriptive models discussed above exist in traditional areas—the representations for events and the inductive techniques differ drastically.

There are two primary approaches to time-series analysis: *regression* methods and *spectral* methods. Regression methods attempt to explain the behavior of a particular variable (the dependent variable, y) in terms of the previous behavior of a set of variables (the independent variables, x_i). If the past behavior of the dependent variable is a function of itself, the system is called *autoregressive.* Regression-based descriptions are the statistical counterparts of the lookback models described above. To fit data to a regression model, the user must specify a particular model, the regression polynominal. Often the form of the regression polynomial is suggested by theory within the field of application. The technique of least-squares regression is applied to estimate the constant parameters of the regression polynomial. If certain assumptions hold, a measure of goodness-of-fit (total explained variance) can be obtained.

Spectral methods attempt to describe the behavior of a particular variable by analyzing its frequency spectrum. This is the continuous frequency counterpart of the discrete periodic models described before. Fourier analysis is used to determine the frequency components that make up the "waveform" of the dependent variable. The independent variable is time.

Here are some examples:

Economic time-series. Let us examine the series

$$S = \langle D_1, D_2, D_3, ..., D_k \rangle$$

where each D_i is an ordered pair, $D_i = (Y_i, X_i)$. Let

Y_i = demand for beef at time i,
X_i = supply of beef at time i.

Economic theory predicts that the demand for beef is a function of the recent values for supply. The form of the regression polynomial is

$$Y_i = B_0 + B_1 X_{i-1} + B_2 X_{i-2}.$$

Using the data in S, the coefficients B_0, B_1, and B_2 can be estimated. The goodness-of-fit of the model can be tested.

Plant management. Imagine a plastics factory where some of the key ingredients are water, oil, and heat. Let

$$S = \langle E_1, E_2, ..., E_n \rangle.$$

Where each $E_i = (y_i, u_i, v_i, w_i)$:

y_i = output per minute of plastic at time i
u_i = input of water (per minute) at time i

v_i = input of petroleum base at time i
w_i = temperature of the reaction chamber at time i.

In order to predict the future production of the plant, we want to describe y_i in terms of previous values of u, v, and w. Water is believed to have a parabolic effect on plastic output. The regression polynomial looks like this:

$$y_i = B_0 + B_1 u_i^2 + B_2 u_i + B_3 v_i + B_4 w_i$$

Using linear regression, we can estimate the coefficients B_0 through B_4 from the data. The regression polynomial need only be linear in the coefficients.

An autoregressive sequence might have the form:

$$y_i = B_0 + B_1 y_{i-1} + B_2 y_{i-2}.$$

Box and Jenkins [3] describe techniques for estimating the degree of autocorrelation (the lookback parameter) from the data. Such techniques permit the researcher to use the data to determine not only the specific content of the model, but also the form of the model. Few such heuristics exist in logical sequential data analysis.

3. THE METHODOLOGY: KNOWLEDGE LAYERS

In this chapter we describe the programming methodology used to develop the Eleusis program. The steps of the methodology are illustrated by indicating how they were applied to Eleusis. The knowledge layer methodology has been very useful in designing the Eleusis program.

3.1 Description of the Methodology

The goal of any programming methodology is to enhance the quality and performance of the program and improve the productivity of the programmer. The knowledge layer methodology seeks to

- simplify the programming process by providing a framework (knowledge layers) for problem decomposition,
- develop general learning programs which are easily adapted to solve related learning problems,
- develop learning programs with sufficient power to solve the problems at hand.

A program designed using the knowledge layer concept is built of distinct layers roughly like an onion (Figure 3).

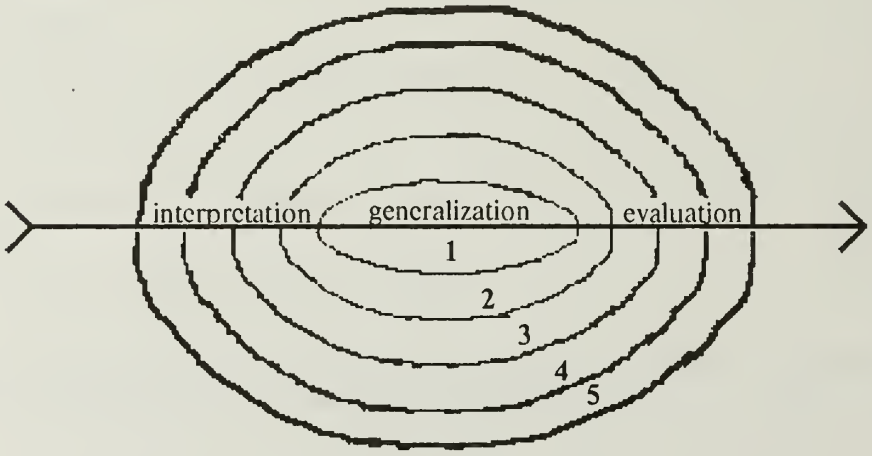


Figure 3. The Knowledge Layer Scheme.

Each layer has access to a specific body of knowledge. Each layer may invoke the next layer within it and may examine the information returned by that layer. The outermost layer interacts with the user of the system to solve a specific class of problems. The innermost layer is the most general. It uses only very general knowledge and algorithms to accomplish its task. The layering is reflected in the generality of the knowledge used at each level, in the scope of variables at each level, and in the flow of control from one level to the next. The knowledge used at each level must all be of the same degree of generality, appropriate to the function of that layer. The variables in that layer can be accessed by outer layers, but not by inner layers. Subroutine calls may only be directed at routines in the current layer or within inner layers. If this discipline is adhered to, the outer layers can easily be removed and replaced by layers better-suited to a particular task.

In order to apply the methodology, it is easiest to proceed by the following steps:

Step 1. Identify the input representations. What kinds of data must the program accept? Do these data contain errors? Are negative examples available? How should the data be described?

Step 2. Identify output representations. What kinds of output descriptions must the program produce? How can these be represented? What description models should be used?

Step 3. Identify the basic algorithms needed to accomplish the learning task. Most learning in non-trivial environments requires three basic operations: interpretation, generalization, and evaluation. Soloway points out [34,35] that incoming data must be interpreted in terms of domain knowledge before they can be generalized. Furthermore, after generalized descriptions have been developed, they must be evaluated to assess their plausibility within the domain in question. This step (step 3) involves determining how the generalization process will take place. A few learning algorithms may be chosen from the many general-purpose algorithms currently in use. Alternatively, new algorithms may be required. These should be designed to use only general knowledge.

Step 4. Identify the transformations required to prepare the input events for the general-purpose algorithms identified in Step 3. This step solves the interpretation portion of the learning problem.

Step 5. Identify the evaluations and transformations necessary to convert the descriptions produced by the general induction algorithms into the desired output descriptions identified in Step 2. This step solves the evaluation portion of the learning problem.

Step 6. Identify the knowledge needed to perform the tasks defined in steps 3, 4, and 5. What knowledge is needed to generalize the events? What knowledge is required to perform the transformations on the input data? What knowledge is required during evaluation? This is a very difficult step to perform because knowledge has a way of entering programs quietly and implicitly. It may help to imagine applying the program to different but related problems.

Step 7. Decompose the program into layers according to the knowledge and tasks performed in each layer. In this step, corresponding functions of interpretation and evaluation are identified and grouped together in layers according to the knowledge required for each function. The layers are designed to surround the basic generalization functions and span the distance from these general-purpose algorithms to the special-purpose problem the program is intended to solve.

3.2 Applying the Methodology to Eleusis

3.2.1 Description of Eleusis

3.2.1.1 Description of the Game. Eleusis was invented over a period of years by Robert Abbott [1,12]. It is an inductive game in which players attempt to discover a secret rule known only to the dealer. The secret rule describes a sequence of cards which are "legal." Players attempt, in their turns, to extend the sequence by playing one or more cards. The sequence of cards which has thus far been played is arranged in a layout (see Figure 4).

mainline:	3H	9S	4C	JD	2C	10D	8H	7H	2C	5H
sidelines:		JD		AH	AS			10H		
		5D		8H	10S			(10S		
				QD				9S <- string played incorrectly		
								4S		
								2S) <- (this card is wrong)		

Figure 4. Sample Eleusis Layout (after [21]).

The layout has a main line which contains all of the correctly played cards in sequence. Incorrect cards are placed in side lines below the main line card which they follow. In a turn, a player may play a string of from one to four cards. If the cards are correct, the dealer places them in the proper positions on the main line. If any one of the cards is incorrect, the entire string is placed on a side line below the last legal card. The string of cards is overlapped so that players examining the layout can recall that only one of the cards in the string need be wrong.

The goal of the game is to get rid of all of one's cards. When a player plays correctly, he or she gets rid of the cards so played. If a player makes errors, the dealer deals additional cards equal in number to double the number of cards played by the player.

The secret rule is invented by the dealer at the start of each round. What prevents the dealer from choosing an impossibly difficult rule? Besides the dealer's natural desire to have an interesting game, the scoring for each round is contrived so that the dealer gets a score equal to the difference between the best and the worst scores for that round. Thus, the dealer is encouraged to choose rules of intermediate difficulty. The rules should stump some players but not others. In this way a large point spread can be created. There are additional rules for the game [1,12], but the above should suffice for the purposes of this thesis.

We wish to construct a program which could aid a human player of Eleusis. This program should

- suggest possible rules to describe the layout,
- evaluate rules suggested by the player, and
- suggest possible cards to play from the player's hand.

Previous work on Eleusis has been done by Barto and Prager [2]. Their work is limited to basic induction tasks. The work shown in [2] was limited to only one rule model—a decomposition model with a lookback parameter of 1.

3.2.1.2 **Typical Rules and Rule Models.** Here are some examples of secret rules (after Abbott[21]):

R1 "If the last card was a spade, play a heart; if last card was a heart, play diamonds; if last was diamond, play clubs; and if last was club, play spades."

R2 "The card played must be one point higher than or one point lower than the last card."

R3 "If the last card was black, play a card higher than or equal to that card; if the last card was red, play lower or equal."

R4 "Play alternating even and odd cards."

R5 "Play strings of cards where each string is one card longer than the previous string and where a string is an ascending sequence of cards starting with an Ace."

R6 "The sum of the values of the last card and the current card must be less than 16."

Where values are mentioned, Ace is usually understood to be 1, Jack 11, Queen 12, and King 13.

The rule models for these rules are precisely the description models introduced in Chapter 2. Rules R1 and R3 are decomposition rules with a lookback parameter of 1. Rule R2 is a DNF lookback rule (a degenerate form of a disjunction) since it expresses the value of one card in terms of the values of the previous card. R5 is also a DNF rule based on segmenting the sequence into strings of ascending cards. R4 is a periodic rule. R6 is a DNF rule, but instead of cards being related by differences, they are related by a sum.

3.2.1.3 Representing Eleusis Rules. Although the VL_1 descriptions introduced in Chapter 2 are sufficient for representing the Eleusis rules described above, it was felt that the user of the Eleusis program would prefer a more elegant and clear representation language. Therefore, VL_{22} was developed as a description language for rules which describe sequences of events. VL_{22} is a successor of VL_{21}[21]. Both languages are subsets of a very extensive description language, VI_2 [21,27].

VL_{22} is an extension of first-order predicate logic which uses a VL_{22} selector as the basic building block for well-formed formulas. VL_{22} selectors are a bit more complex than VL_1 selectors:

[*function* (*variable-list*) *relation function* (*variable-list*) *operation value-list*]

Variables in the variable-lists refer to specific cards or strings. The same subscripting convention used in VL_1 is used in VL_{22} to indicate the order of the cards. For example, card0 refers to the current card; card1, to the card before card0; etc. Functions applied to these variables take on values from explicitly defined domains (exactly like VL_1 variables). Difference and sum variables are not needed in VL_{22} since functions can (optionally) appear in the reference. The operations required to express Eleusis rules are plus, minus, and plus-or-minus. Each VL_{22} expression is assumed to be universally quantified over the entire event sequence (with the implicit condition that card0 is adjacent to card1, card1 to card2, etc.). Table 5 shows the VL_{22} equivalents of the Eleusis rules listed above. Note that the dummy variable string is used to describe a string of cards in a segmented rule. Subscripts are applied the strings as well as to cards.

Table 5. VL_{22} Descriptions of Eleusis Rules.

R1	[suit(card0) = suit(card1) + 1]
R2	[value(card0) = value(card1) + -1]
R3	[suit(card1) = black] = > [value(card0)> = value(card1)] V [suit(card1) = red] = > [value(card0)< = value(card1)]
R4	Period ([valuemod2(card0) = even], [valuemod2(card0) = odd])
R5	string = [value(card0) = value(card1) + 1] : [length(string0) = length(string1) + 1]
R6	[value(card0) < = - value(card1) + 16]

3.2.1.4 Plausible Rules. In order to discover Eleusis secret rules, we must first define what we are looking for. Induction is the process of selecting plausible descriptions from the space of all possible descriptions. In Eleusis, we are searching for plausible rules to describe the layout. Abbott gives some guidelines for forming good Eleusis rules, and these quidelines can be used to define characteristics of plausible rules.

First of all, conceptual simplicity is important. Complex Eleusis rules will not score well for the dealer because no one will be able to guess them. Even apparently trivial rules are quite difficult for people to guess. Secondly, some rules permit many cards to be legal at many points. Abbott observes that rules which, on the average, permit fewer than one-fourth of the deck to be played are

usually easier to discover than rules which typically allow half the cards to be played. A rule which permits any card to be played any time is quite difficult to discover because no negative examples are ever produced. Thirdly, most dealers arrange the rule so that every card is playable at some time during the game.

These plausibility constraints can be used to evaluate rules produced by the general induction algorithms. To measure conceptual complexity, we can count the number of selectors in the rule. Other syntactic measurements, such as measuring the number of values in a reference, can be used to approximate conceptual complexity. The size of the set of legal cards can be deduced from the VL_1 description of the rule. An estimate of the average size of the set of legal cards can be developed and used to test the plausibility of the rule. It is relatively easy to determine that all cards are playable at some point or that the rule has no dead ends.

One comment concerning the models and plausibilities for Eleusis rules is important. The computational tool for Eleusis is designed to assist a human player who is playing Eleusis with other human players. If all players had this tool available, the types of rules that could be played would undoubtedly change. Firstly, rule models which the Eleusis tool cannot discover would be played very often. Secondly, the secret rules would tend to become more complex, since, with the help of the computational tool, the standard types of rules would be much easier to discover. The present tool is not directed at overcoming these problems. I do not even claim that the Eleusis tool described here will discover the full range of Eleusis rules used in ordinary human play. This Eleusis program is capable of fitting data to the decomposition, periodic, and DNF models and of evaluating the quality of the fit based on knowledge of how humans play the game Eleusis—nothing more.

Eleusis rules other than those which can be discovered using the Eleusis program have occurred in games played by the author. These include rules which use segmentations based on position rather than card values and rules involving existential quantifiers. For example, the rule "Segment the layout into pairs of cards so that cards 1 and 2, 3 and 4, 5 and 6, etc., make up the segments. The derived sequence is formed by summing the values of the two cards in each segment. The rule is that segments with odd and even sums must strictly alternate." This fits a periodic model, but the segmentation cannot be discovered by the current program.

3.2.2 **Design Steps for an Eleusis Tool.** Here are the steps of the design methodology applied to the design of the Eleusis program:

Step 1. Identify Input Representations. The input representations for the Eleusis program are symbols of the form '2C' or 'JD' representing cards in a card deck. The input is entered in order of play. Each string of cards (one to four cards in length) is entered using a CARD command along with the judgment of the dealer:

```
CARD 2C 3D:Y;
```

This command indicates that a player played two cards and the dealer pronounced them correct. The input is stored in the program as a linked list in the form of the layout.

Step 2. Identify Output Representations. Rules produced by the program are written in VL_{22} as described above. The rules fit the three description models described in Chapter 2: periodic, decomposition, and DNF.

Step 3. Identify Generalization Algorithms. The three algorithms presented in Chapter 2 are the algorithms used in the inner-most layer of the Eleusis system. Each algorithm is designed to fit unordered VL_1 events to one of the description models. Each algorithm produces a description in the form of a disjunction of VL_1 complexes.

Step 4. Identify Interpretation Steps. Four interpretation steps can be identified. The first step is to convert cards to canonical VL_1 complexes containing the suit and value of each card. Thus, 2C becomes [suit = clubs][value = 2].

The second step is to derive additional variables which may lead to plausible descriptions of the layout. Color and valuemod2 (the value of the card modulo 2) might be added at this point. Also, some indication of whether the card is a faced card or has a value which is a prime number might be desired. In the second step we could transform [suit = clubs][value = 2] to [suit = clubs][value = 2][color = black][valuemod2 = 0] [faced = false][prime = true].

The third step involves segmenting the layout. As discussed in chapter 2, many interesting and plausible descriptions are based on segmentation. In Eleusis, we segment the layout into strings of maximal length which satisfy a segmentation condition. Although it might be possible to develop some techniques for inferring the segmentation condition from the data, we have chosen to use a hypothesize and test approach. The user provides a list of segmentation conditions. The program attempts to segment the layout with each condition and then evaluate how plausible the segmentation is. For example, if the segmented layout has nearly the same number of events as the original layout, then it is very unlikely that the layout is well-described using that segmentation condition. Conversely, if the whole layout satisfies the segmentation condition so that only one segmented event is produced then this is not a plausible segmentation either (invariant properties of the entire layout are discovered by other procedures).

The final transformation step involves making the order of the events explicit in the events and removing the order from the sequence. Once a model and a value for the lookback parameter have been chosen, it is easy to develop events like those of Table 1 which contain descriptions of the current card, the preceding cards, and relationships between them. Thus, this step computes sum and difference variables.

Once the events have been processed by these transformation steps, they are ready to be generalized using the VL_1 induction algorithms of the inner-most layer.

Step 5. Identify Evaluation Steps. Three evaluation steps can be identified. The first step examines rules developed by the VL_1 induction algorithms and filters them to remove redundant information. For example, it often happens that the VL_1 induction algorithms develop descriptions like:

[face1 = false] => [value0> = J][value0>value1][face0 = true] V
[face1 = true] => [value0< = 10][value0<value1][face0 = false]

The selectors [value0>value1] and [value0<value1] are redundant, but logically correct, statements. The VL_1 induction algorithms cannot remove these selectors since the algorithms are not aware of the order of the events. The outer layers can use knowledge of the order of events to remove these selectors.

The second evaluation step is required when the layout has been segmented. Using a segmentation condition, the end of the layout cannot be successfully segmented. For example, if we had the sequence:

$$S = \langle 3, 4, 4, 5, 5, 5, 6, 6, 6, 6, 7, 7 \rangle$$

we would not want to create an event for the sevens. Such an event would indicate that there was a string of sevens of length 2. If the VL_1 induction algorithms received such an event, they would not be able to discover that the length of a string always increases by 1. Thus, the segmentation process must always leave the end of the layout unsegmented. However, when a

description is developed, it might in fact be inconsistent with the unsegmented portion of the layout. If the sequence had looked like

$$S = \langle 3, 4, 4, 5, 5, 5, 6, 6, 6, 6, 7, 7, 7, 7, 7, 7, 7 \rangle$$

then the VL_1 induction algorithms would incorrectly describe the sequence. Each description produced by the VL_1 induction algorithms must be checked to verify that it is consistent with the tail end of the layout.

The third evaluation step involves assessing the plausibility of the descriptions in terms of Eleusis. The complexity of each description must be measured (approximately). The average size of the set of legal cards must be measured in accordance with the plausibility criteria mentioned above. It must not be possible to reach a dead-end while playing according to the rule. Lastly, the description must be checked to see that it is consistent with negative string plays. Recall that, in Eleusis, if a player plays a string of cards (2, 3, or 4 cards), and any one of the cards is in error, the entire string is placed on a sideline below the main line. Although this information is difficult to use during rule discovery, it is necessary to check each description developed by the VL_1 induction algorithms to see that it is consistent with these negative string plays. At least one of the cards in each negative string must be illegal according to the description.

Once the descriptions have passed through all of these evaluation steps, they must be converted to VL_{22}. In the Eleusis program, the discovered rules are maintained in a rule base along with rules which the user may have entered into the system. The rule base is consulted when the player wants to play a card.

Step 6. Identify Knowledge Requirements. First, we list the knowledge required for the interpretation steps, then for the generalization step, and finally for the evaluation steps. The first interpretation step (converting 2C to a VL_1 complex) merely requires knowledge of the card notation. The second step (adding color, valuemod2, etc. to the events) requires knowledge of the definitions of the added variables. The user is able, in the Eleusis program, to enter the definition of a new variable as a VL_{22} complex. For example, color can be entered as:

```
DEFINE COLOR = RED   [suit(card0) = hearts, diamonds],
               BLACK [suit(card0) = spades, clubs];
```

The segmentation process requires knowledge of the ordering of the layout. The program must know how to compute the difference variables between adjacent events in order to determine that they satisfy the segmentation condition. This in turn requires knowledge of the domains of the variables. The segmentation process must also know how to segment negative events properly. Violation of either the segmentation condition or the segmented rule can cause a card to be illegal. However, at the time the layout is being segmented in preparation for rule discovery, only the segmentation condition is known.

The last interpretation step requires knowledge of the ordering of the layout so that the unordered events may be developed. Knowledge of how to compute sum and difference variables is obviously needed. This in turn requires knowledge of the domains and domain types of the variables. The last interpretation step prepares events for a specific model with a specific lookback parameter, so this information must be available.

The generalization steps require knowledge of the domains and domain types of the variables. In particular, the domain type-specific rules of generalization must be available during the generalization process. The decomposition algorithm requires knowledge of which variables are left-hand side variables. The algorithms also have a good deal of knowledge available in

their cost functionals. The cost functions must measure the plausibility of the descriptions under development.

Knowledge required to evaluate the rules and remove irrelevant variables includes knowledge of the ordering of the events and knowledge of each rule model. The process which removes redundant difference variables must understand the relationship between the difference variables and the variables from which they were derived.

Knowledge required to test the tail end of the segmented layout for consistency is precisely identical to the knowledge required to initially segment the layout.

Knowledge required to estimate the average size of the set of legal cards includes knowledge of the relationships between variables and knowledge of how segmentation interacts with the description models. The process of verifying that each negative string play contains a bad card requires little knowledge beyond the knowledge of how negative string plays are handled in Eleusis. The conversion of a VL_1 rule to a VL_{22} rule is a straightforward syntactic manipulation.

Step 7. Decompose the System into Layers. Figure 5 indicates the layers of the Eleusis system and the functions of each layer.

Layer	Function	Knowledge Used
5	User Interface	Cards, VL_{22} syntax
4	Eleusis	Color, valuemod2, negative strings, plausible Eleusis rules
3	Segmentation	Ability to segment layout, check tail end of segment, plausible segmentations
2	Sequential Analysis	Models and parameters, ordering in the layout, sum and difference variables
1	Basic Induction	Domains and domain types, basic algorithms, cost functions

Figure 5. Architecture of the Eleusis System.

The top-most layer (Layer 5) provides the user interface. It also performs the first interpretation step by converting cards into VL_1 complexes.

Layer 4 contains all Eleusis-specific knowledge including all of the knowledge of the relationships between variables and knowledge of negative string plays. This layer performs the second interpretation step by expanding the input events to contain all possible variables which might be relevant. This layer removes negative string plays from the layout and uses them later to evaluate the descriptions returned from layer 3. Layer 4 also converts VL_1 descriptions to VL_{22}. Knowledge of plausibility in Eleusis is used in this layer to evaluate descriptions according to the average number of cards playable under the rule.

Layer 3 performs all functions relating to segmentation. It segments the layout according to a list of possible segmentation conditions and evaluates each. Those which it finds to be promising it hands to layer 2 for further discovery. It evaluates descriptions returned from layer 2 to guarantee that they are consistent with the tail end of the layout.

Layer 2 performs the function of removing order from the layout. It computes the unordered

VL_1 events including the sum and difference variables. For each model, it develops a specific set of events and passes them to layer 1 for generalization. Layer 2 filters the resulting descriptions to remove redundant selectors.

Layer 1 performs the basic generalization tasks described in Chapter 2. It implements the three VL_1 induction algorithms discussed above.

3.2.3 **Other Functions of the Eleusis Program.** In addition to discovering plausible Eleusis rules, the Eleusis program also provides other valuable services to its user. First, it permits the user to enter Eleusis rules in VL_{22} form. It checks those rules for consistency with the layout and adds them to its VL_{22} rule base. Second, it permits the user to enter the cards that s/he has in her or his hand. When the user issues an EVALUATE command, each rule in the VL_{22} rule base is processed against the layout to determine which cards are currently legal according to that rule. Each card in the player's hand which is currently legal is marked. The user can display this information in order to choose a card to play, or s/he can ask the system to suggest a card to play. The system plays according to two strategies. The conservative strategy is to play a card which is legal under as many rules as possible. The discriminant strategy is to play a card which will eliminate roughly half of the rules from further consideration.

These additional functions require two major additions to the Eleusis program. Besides rule discovery, the program needs the ability to check a rule against the layout to determine if it covers all of the mainline and is consistent with all of the negative examples on the sidelines. This is called the Critic function. The program also needs the ability to determine which cards are legal extensions of the layout according to a given rule. This is called the Performance Element function. These two functions are discussed below in connection with the learning system model proposed by Buchanan, *et al.* [7].

3.3 Comparison of the Knowledge Layered System With Other AI Systems

The layered methodology described here is quite different from many traditional AI approaches. In fact, knowledge layers owe more to structured programming ideas than to traditional AI programming techniques such as heterarchy or production systems.

The strict layered style, where each layer invokes the layer below it, is a very hierarchical system. Knowledge is used only at one level. Widely different types of knowledge are not intermixed. The knowledge discipline does not, for example, permit layer 1 to use knowledge of the specific segmentation performed at layer 3. Both heterarchy and production systems, on the other hand, permit any piece of knowledge to be used whenever possible. Heterarchical systems were originally developed to bring semantic knowledge to bear on syntactic problems in natural language (parsing [22]), vision (line finding [11]), and speech (phone identification [10]). These systems have distributed control and a large, highly integrated knowledge base. The Eleusis system, on the other hand, has hierarchical control and a set of largely decoupled knowledge bases. The generality (and adaptability) of the Eleusis program is based on the fact that the outer layers of the onion may be peeled off and replaced by other layers without requiring any changes in the remaining layers. This is not possible with heterarchical systems.

Much of the knowledge used in the Eleusis system is procedural and not easily modified or augmented. Entire layers can be removed and replaced without too much difficulty, since the interfaces between layers are narrow and well-defined. But small changes, such as those possible by adding or deleting a rule from a production system, are not easily accomplished because the "blocks" of knowledge are much larger (entire layers). The program lacks the incremental enhancement capability of production systems, but by removing and replacing entire layers, the program can achieve greater range of application than a single, specific production system.

The AM system developed by Lenat [22] is essentially the opposite of the knowledge layer

approach. Not only does it operate by distributed control with a modified production system, but it builds an elaborate frame system of concepts rather than a simple, closed expression describing some object.

Two of the programs most closely related to the present work are Soloway's BASEBALL system [34,35] and Larson's INDUCE-1 program [20,21,27]. Soloway's program was provided with low level snapshots of a baseball game and asked to induce some descriptions of the game given its knowledge of sports and competitive games. The program first transforms the low level descriptions in several ways and then generalizes and evaluates the results. In order, there are the following layers of processing:

1. Filtering out periods of little activity (justified by general attention heuristics)

2. Segmenting the stream of events into activity cycles (justified by game heuristics)

3. Embellishing simple activities with knowledge of common sense physics (causes, enabling conditions, etc.). The resulting events are called action schemas.

4. Hypothesizing goals and relationships (using knowledge of team membership and competition, attach goals to the actions of each player). The resulting events are called Causal-Link Schemas.

5. Extracting the final goal of each Causal-Link Schema.

6. Generalizing the Causal-Link Schemas to develop classes of episodes (e.g. walk, single, double). The generalization process makes use of game-specific knowledge to determine how much to generalize each event.

7. Generalizing the final goals extracted at step 5 to obtain classes of final competitive goals (e.g. out, hit).

8. Evaluating the resulting descriptions for internal consistency and predictive power.

The use of layers of interpretation, each with special knowledge appropriate to that layer is very similar to the system described in this thesis. Note, however, that the BASEBALL system proceeds from general-to-specific. The outer-most layers apply the most general heuristics for transforming the input data. The inner-most layers have detailed knowledge of competitive games, plan structures, and appropriate techniques for generalization. The generalization steps are the least general part of the system. While in the Eleusis program, the layers serve to connect a general tool with a specific domain, in BASEBALL domain knowledge is used most extensively in the heart of the system.

Another difference between the present work and Soloway's BASEBALL program involves description evaluation. Soloway includes an extra layer for description evaluation. The knowledge-layer approach advocates that the descriptions, after being induced, should traverse all of the layers on their way out to the outermost layer. Each layer has an opportunity to evaluate the descriptions in terms of the knowledge available at that layer.

Another system which is similar to the Eleusis program is the INDUCE-1 program developed by Larson. This program has two main levels. The outer-most level uses the representation language VL_{21} to describe structured objects such as toy block structures, trains, and biological cells. The outer level conducts a search to find the most general description of one class of objects which covers no negative examples (no members of other classes of objects). The search attempts to determine which variables and structures are most relevant to discriminating one class from another. Using a chosen structure, the VL_{21} induction problem can then be converted to a VL_1 problem. The Λ^q algorithm is used, in the inner layer, to solve this simple VL_1 problem and the result is

returned to the outer layer where it is evaluated and eventually printed to the user. INDUCE-1 is a good example of the knowledge-layer design. The outer layer uses knowledge about structured descriptions; the inner layer is more general and uses only knowledge about VL_1 complexes. INDUCE-1 is a general program upon which other layers may be built.

Most prior work in sequential data analysis has sought to induce plausible grammars (or equivalently, automata) which could generate or extrapolate a sequence of events [17,40]. Grammars have advantages:

- Grammars provide a natural representation for segmented descriptions. A particular grammar rule can be used recursively by a new grammar rule. This solves the segmentation problem.

- Grammars are well-understood mathematically.

Grammars also possess disadvantages which make them difficult to use for describing sequential event sets:

- Grammars describe sequences of events by generating them. It is difficult to write a grammar which merely constrains the possibilities at a point. For example, to write a grammar which permits the next event to have an even value, we must write:

  even -> 2 | 4 | 6 | 8 | 10 | 12
  S -> even | S even

  Furthermore, in order to extend a series, the start symbol, S, must be reduced to terminal symbols. All possible extensions of a series must be generated in order to develop a prediction.

- Grammars do not correspond to the way people describe sequential events. The grammar above describes a sequence of even numbers. I think people tend to describe such a sequence logically as

  $$\forall x \ \text{even}(x) \qquad x \text{ in the sequence.}$$

  It is important that a computational tool produce descriptions which are conceptually simple and in accordance with human-based descriptions. In the game Eleusis, typical rules are much more easily expressed in logic than as a grammar.

- Grammars lack many useful operations. Unadulterated grammars use only juxtaposition. Even in the augmented grammars used in general production systems, juxtaposition plays a major role. Yet a good description of a sequence of events is event-centered. The characteristics of the next event are described in terms of its immediate environment. For example, in R1 (Table 5) if the previous card is a club, we must play a diamond. In R4, the position of the next card in the layout determines whether it must be odd or even. These event-centered descriptions are very clear, and they make it very easy to compute legal extensions of the sequence. Such descriptions have grammatical counterparts, but these counterparts are rarely as succinct and clear.

3.4 Relationship to the Learning System Model

The learning system (LS) model proposed by Buchanan, *et al.* [7] has influenced the design of this program for Eleusis. According to Buchanan, a learning system contains 6 components: an instance selector (IS), a learning element (LE), a performance element (PE), a critic (CR), a blackboard (BB), and a world model (WM). The learning element is responsible for developing descriptions from examples; the instance selector selects the examples to be submitted to the LE; the performance element uses the descriptions developed by the LE to perform some task; and the critic criticizes the activities of the performance element and suggests improvements to the LE. The blackboard is a central knowledge base through which the various learning system components communicate. The world model contains (implicitly or preferably explicitly) knowledge about the problem domain which is assumed and used by the learning system. Buchanan points out a second role for the critic. It may serve to evaluate intermediate results from the LE and guide its search for plausible descriptions.

Layered learning systems are also described by Buchanan. When one LS is layered on top of another, the lower LS serves as the PE for the upper LS (See Figure 6).

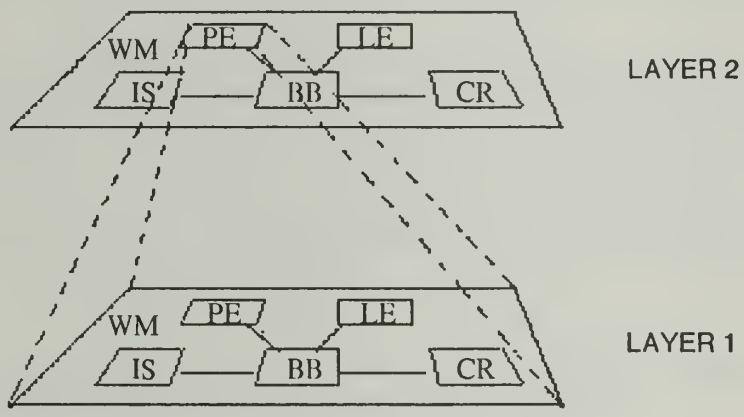


Figure 6. Layered Learning Systems (After Buchanan, *et al.* [7])

The upper LS may change the WM of the lower LS and thus improve its overall performance. The layers communicate through the BB.

The Eleusis tool can be viewed as a layered LS—but with significant departures from the Buchanan model. In the Eleusis system, the learning element of each lower layer is contained in (and called by) the learning element of the layer above it (see Figure 7). Similarly, the PE and CR are each contained in their counterparts in the layers above. Each layer does modify the WM of the layer below it. But the modifications are not as major as those suggested by the LS model (e.g. changing heuristics, modifying the program). Each layer does not view the layer below it as a system to be evaluated and improved. Rather the different layers are distinguished by the type of knowledge which they use, and they work together to accomplish a learning task.

In LS terminology, layer 4 of the Eleusis program is the outermost LS. Layer 4 adds relevant variables to the sequence of VL_1 events and calls the LE of layer 3. The LE of layer 3 segments the input sequence of events and passes the resulting segmented sequence to the LE of layer 2. Layer 3 defines some portions of the WM of layer 2 such as the variables and events which are to be processed by layer 2. It sets parameters (lookback parameters and limits on the suggested lengths of periods) and provides advice concerning which models to use.

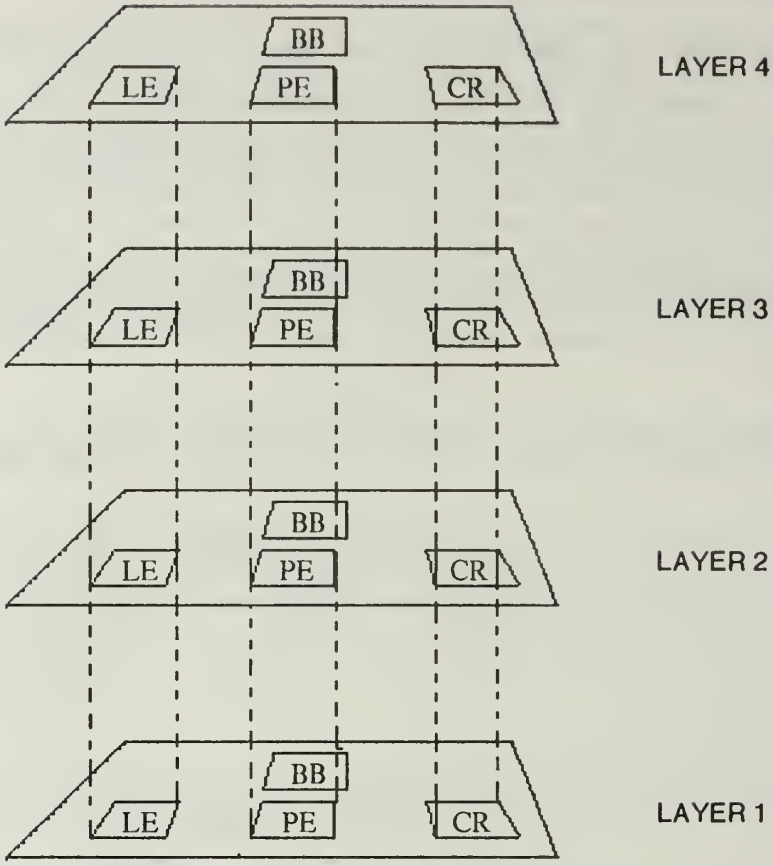


Figure 7. Eleusis as a Layered Learning System.

Similarly, the LE of layer 2 removes order from the sequence, derives new unordered events, and calls the LE of layer 1 to fit a particular set of events to a particular model with a particular lookback parameter.

When the LE of layer 1 has developed a description, it returns it to layer 2. Layer 2 can evaluate the description in terms of its knowledge of the order of events. The more plausible descriptions developed in layer 2 are returned to layer 3 where they can be evaluated using segmentation knowledge.

As noted above, layer 4 conducts extensive tests of plausibility on each rule returned from layer 3. Rules must be checked against negative string plays, evaluated to determine the size of the set of legal cards, and then converted to VL_{22}.

The PE and CR of the Eleusis system behave in similar ways. The Eleusis critic merely evaluates a rule to determine if it is consistent with the layout. The CR of layer 4 follows the same event derivation process as the layer 4 LE. It must also convert to VL_1 the VL_{22} rule that is being checked. The CR of layer 3 segments the layout according to the segmentation condition in the rule (if any)) and calls the CR of layer 2. The CR of layer 2 derives the appropriate unordered VL_1 events according to the model and lookback of the rule and then calls layer 1. Layer 1 merely checks to see that the rule describes all positive events and no negative events. The result is passed up the levels. Layer 3 must check the tail end of the layout to guarantee that it satisfies the rule. Layer 4 must verify that each negative string play contains at least one illegal card. The result of

the rule evaluation is finally returned to layer 5 and printed to the user. The CR is designed to give a yes-or-no answer to the question: is the description consistent with this sequence of events. It does not provide advice to the LE or assign blame to particular parts of the description.

The performance element of Eleusis is charged with the task of determining which cards are currently playable according to a given rule and a given layout. The PE of layer 4 again adds color, valuemod2, and so forth to the events, converts the rule to VL_1, and calls the layer 3 PE. The layer 3 PE segments to layout according to the rule and calls the layer 2 PE. The layer 2 PE determines which complexes in the rule are presently applicable. For a periodic rule, it determines what the next phase must be and returns a complex which describes that phase. For decomposition and DNF rules, the layer 2 PE must examine previous events to determine which alternatives in the rule are presently applicable. There is no layer 1 PE since layer 1 has no knowledge of the ordering of the events and therefore cannot know how to extend the sequence. The layer 2 PE returns a disjunction of complexes which predict the next card to layer 3. The layer 3 PE processes these conjuncts according to the segmentation condition and returns a set of legal cards to layer 4. Layer 4 has no PE function and merely returns the set of legal cards to layer 5. (Note that this is a slight violation of the knowledge discipline. The layer 3 PE has some knowledge of playing cards. The program should be improved so that this knowledge is not necessary. The improvements require more general deduction methods at layer 3).

The LE in Eleusis differs from the Buchanan LS model because it contains three separate functions: interpretation, generalization, and evaluation. In this system, the learning element of each layer contains an interpretation step, a generalization step, and an evaluation step. Interpretation involves the process of what Michalski calls *constructive induction* [9,27]. New variables are added to, and old variables are removed from the original events to develop new *derived events.* Each layer performs constructive induction on the events it receives, and it passes the derived events to the next lower layer. Constructive induction is always a knowledge-based process by which events are transformed to more appropriate representations.

The Eleusis program is thus a layered LS in which the learning elements, performance elements, and critics of each layer cooperate with the layers above and below.

4. EVALUATION OF PROGRAM PERFORMANCE.

In this chapter, the performance of the Eleusis program is evaluated by presenting examples of its execution. Possible improvements and extensions to the program are also described.

4.1 The Implementation

The Eleusis program is written in Pascal for the CYBER 175 (Control Data Corporation). The program is roughly 9500 lines in length and occupies 128K words (60 bits per word) when running non-trivial examples. Of that 128K, 50K is code and static data, and the remainder is dynamic data. The program does not implement all of the ideas discussed in this thesis. In particular, the following features remain unimplemented:

- Level 4 Eleusis Plausibility. The level 4 subroutine which is intended to estimate the average size of the set of legal cards under a given rule is unimplemented. Thus, the program often prints unintelligent, implausible rules.

- Level 3 Segmentation Check. In the LE and the CR, the program does not check the tail end of the segmented layout to see that it is consistent with the rule in question. Thus, the system can induce rules which, although they are consistent with the segmented portion of the layout, are inconsistent with the very last few cards. The PE does check these cards.

- Level 2 Plausibility. At level 2, no attempt is made to filter out redundant selectors.

Aside from these subroutines, the program completely implements the ideas mentioned in this thesis.

4.2 Sample Runs

The example runs in this section are based on actual games played by the author (with his "research associates" at *Eleusis parties*!). Although knowledge of these games assisted the development of the Eleusis system, they were not used during program development and testing. Each example was run with the same parameter settings (except as noted). It is intended that the program be run with a standard, relatively conservative, set of parameters. If the user of the program is dissatisfied with the results obtained using those parameters, then they may be increased. Ideally, the program would make such decisions based on knowledge of what constitutes good Eleusis rules. It would be very nice, for example, if the system demonstrated a satisficing behavior. It could examine the simplest (and computationally cheapest) possibilities first and then move on to more complex description possibilities if the simple ones did not work. The system would stop searching as soon as it found a few plausible rules.

However, at present, the user of the program indicates a space of possibilities by setting parameters and the program searches that space and returns all plausible rules found. For these examples, the space of possibilities involved the decomposition and periodic description models only. For decomposition, the lookback parameter was set to one. For periodic rules, the number of phases was set to one or two, and the lookback between phases was set to zero or one. Five segmentation conditions were given to the program to investigate. These are:

[value(card0) = value(card1)]
[suit(card0) = suit(card1)]
[value(card0) = value(card1) + 1]
[valuemod2(card0) = valuemod2(card1)]
[color(card0) = color(card1)].

For segmented rules, the system was told to investigate only a degenerate form of the periodic model. This degenerate period has a lookback of one and only one phase. Such a degenerate periodic description can be used when a single conjunctive description of the layout is desired. The DNF model could have been used to achieve the same effect, but the periodic algorithm is more efficient than the A^q algorithm. The program was provided with rules to generate the following relevant derived descriptors:

a. Color. Color of the card.

b. Face. True if the card is a faced (picture) card, false otherwise.

c. Prime. True if the card has a prime value, false otherwise.

d. Mod2. Takes the value 0 if the card is even valued, 1 otherwise.

e. Mod3. Takes on the value of the card modulo three. This is an example of a "noise" descriptor since it is very unlikely that it will be involved in any plausible descriptions.

f. Lenmod2. Takes on the value of the length of a subsequence, modulo 2.

4.2.1 Example 1. Below is the layout for the first example rule. It is a very simple rule and the program discovers three equivalent descriptions for it:

```
JC  4D  QH  3S  QD  9H  QC  7H  QD  9D  QC  3H  KH  4C  KD  6C
KC  5S              4S      10D
    7S

6C  JD  8D  JH  7C  JD  7H  JH  6H  KD   (<- main line continued)
```

The program discovered the following descriptions of this layout (494 milliseconds were required):

```
RULE 1: LOOKBACK: 1 NPHASES: 0 DECOMP
[FACE(CARD1) = FALSE] => [VALUE(CARD0) >= JACK]
                         [VALUE(CARD0) >VALUE(CARD1)]
                         [FACE(CARD0) = TRUE] V
[FACE(CARD1) = TRUE] => [VALUE(CARD0) = 3..9]
                        [VALUE(CARD0) <VALUE(CARD1)]
                        [FACE(CARD0) = FALSE]

RULE 2: LOOKBACK: 1 NPHASES: 1 PERIODIC
PERIOD([VALUE(CARD0)>= 3][VALUE(CARD0) <>VALUE(CARD1)]
       [FACE(CARD0) <>FACE(CARD1)])

RULE 3: LOOKBACK: 1 NPHASES: 2 PERIODIC
PERIOD([VALUE(CARD0)>= JACK] [VALUE(CARD0)>= -VALUE(CARD1) + 20]
       [FACE(CARD0) = TRUE],
       [VALUE(CARD0) = 3..9] [VALUE(CARD0) = -VALUE(CARD1) + 5..14]
       [FACE(CARD0) = FALSE])
```

Rule 1 expresses the rule as a decomposition rule with a lookback of 1. Most periodic rules which have disjoint phases can be expressed as decomposition rules. Rule 2 expresses the rule as a single conjunction. This is possible because face vs. non-face is a binary condition and there are precisely two phases to the rule. Rule 3 expresses the rule in the "natural" way as a periodic rule of length 2.

Notice that, although the program has the gist of the rule, it has discovered a number of redundant conditions. For example, in rule 1, the program is not able to use knowledge of the fact that [value(card0)>=jack] is equivalent to [face(card0)=true] to remove the former selector. Similarly, because of the interaction of the two conditions, [value(card0)>value(card1)] is completely redundant. However, it is not difficult for the user of the program to ignore these irrelevancies in this case. We shall see more extreme examples of this problem below.

With these rules, we can demonstrate the performance element of the Eleusis system. Assume that the user of the program is actively playing an Eleusis game and has entered his/her hand into the program. Then the user can invoke the EVALUATE command to evaluate each rule according to the cards in the hand. When the player lists the hand, each card is listed along with the rules under which it is currently playable:

```
CONTENTS OF HAND:
RULE:  1   2   3   4   5   6   7   8   9   10  11  12  13  14  15
CARD:
 KC
 QS
 JD
 10H       Y
 9C    Y   Y   Y
 8S    Y   Y   Y
 7H    Y   Y   Y
 6D    Y   Y   Y
 5C    Y   Y   Y
 4S    Y   Y   Y
 3H    Y   Y   Y
 2D
 AC
 3S    Y   Y   Y
 7H    Y   Y   Y
 5C    Y   Y   Y
```

The letter 'Y' indicates that the card listed on the left is legal according to the rule numbered by the number of the column. Notice that the program believes that we cannot play either aces or twos. If we ask the program which card to play (via the PLAY command), it will select the 9C under the conservative strategy, and the 10H under the discriminant strategy.

4.2.2 Example 2. This example shows what happens when the phases of a period are not strictly disjoint. Recall that the program seeks symmetrical, disjoint descriptions for the phases of a period and for the if-then cases of a decomposition rule. The rule intended by the dealer was "play a periodic rule where the first phase may be either a spade or a heart, and the second phase may be either a diamond or a heart." The layout for the game was:

```
9S  4D  KH  3D  KS  5D  AS  2D  KH  6H  QS  AH  AH  10D 7S  7H
    9C  4C  5C  QS                  6S              JC
    7C                                              JD
```

The program discovered the following rules using the decomposition and periodic rule models:

```
RULE 1: LOOKBACK: 0 NPHASES: 1 PERIODIC
STRING = [MOD2(CARD0) = MOD2(CARD1)] : PERIOD([LENGTH(STRING0) = 1,2,5])

RULE 2: LOOKBACK: 1 NPHASES: 2 PERIODIC
PERIOD([VALUE(CARD0) = VALUE(CARD1) + -0..12]
       [VALUE(CARD0) > = -VALUE(CARD1) + 6]
       [SUIT(CARD0) = HEARTS..SPADES]
       [SUIT(CARD0) = SUIT(CARD1) + 3..1]
       [MOD3(CARD0) = 0..1][MOD3(CARD0) = -MOD3(CARD1) + 1..2],

       [VALUE(CARD0) < = 10]
       [VALUE(CARD0) <>VALUE(CARD1)]
       [VALUE(CARD0) = -VALUE(CARD1) + 5..15]
       [SUIT(CARD0) = DIAMONDS..HEARTS]
       [SUIT(CARD0) = SUIT(CARD1) + 3..1][COLOR(CARD0) = RED]
       [COLOR(CARD0) = COLOR(CARD1)][FACE(CARD0) = FALSE]
       [FACE(CARD0) = FACE(CARD1)][MOD3(CARD0) = -MOD3(CARD1) + 1..2])
```

The first rule is absolutely miserable. Because the plausibility evaluation part of the program is only partially implemented, this rule manages to make its way up to the top level. The rule says that the main line is made up of strings of cards which have the same value modulo 2. These strings are either 1, 2, or 5, cards in length. Under the Eleusis knowledge of plausibility, this rule would be eliminated because there are many times when any card is legal.

The second rule is not much better. One can see that the dealer's rule was discovered (e.g. [suit(card0) = diamonds..hearts]), but when the periodic algorithm attempted to remove overlapping selectors, it removed the significant selectors along with the insignificant ones. Recall that the algorithm backs up in such cases and returns the ungeneralized rule.

Since these descriptions were so bad, the program was instructed to examine a DNF model for this game. The following rule was discovered in 5189 milliseconds:

```
RULE 3: LOOKBACK: 1 NPHASES: 0 DNF
[VALUE(CARD0) < = -VALUE(CARD1) + 16][SUIT(CARD0) = DIAMONDS..SPADES] V
[SUIT(CARD0) = HEARTS]
```

This rule states that hearts are always legal, and that if the sum of the values of the current card and the previous card is less than or equal to 16, then the current card may be a diamond or spade. Although this rule is incorrect, it does serve the useful purpose of isolating the relevant variables. A user of the program might then be able to identify the rule.

It is clear that the program does not handle asymmetrical rules well. The DNF model is able to isolate relevant variables even though the rule it discovered will lead to incorrect play.

4.2.3 Example 3. In this example, we show the program discovering a segmented rule. Notice that in the previous rules, although several segmentation conditions were suggested, only one (very poor) segmented rule was discovered. Included in the parameters for these example sessions are the plausibility limits for segmentation. These were set so that a segmentation of the layout must produce at least 5 segments, and the number of events in the segmented layout must be no more than half the number in the original layout. These plausibility limits have been very successful in weeding out unpromising segmentations. Furthermore, in all of our testing of the program, never has a segmentation condition been erroneously eliminated from further consideration.

The layout for this example is:

```
AH  7C  6C  9S  10H 7H  10D JC  AD  4H  8D  7C  9S  10C KS  2C  10S
        KD          5S      QD      3S          9H      QH
        JH                                      6H      AD

2C  10S JS  AS  5C  KC   (main line continued)
```

The program only discovered one rule for this layout, precisely the rule which the dealer had in mind (1221 milliseconds required):

```
RULE 1: LOOKBACK: 0 NPHASES: 1 PERIODIC
STRING = [COLOR(CARD0) = COLOR(CARD1)]: PERIOD([LENMOD2(STRING0) = 1])
```

The rule states that one must play strings of cards with the same color. The strings must always have odd length. Actually, the rule which the dealer had in mind had one additional constraint: a queen must not be played adjacent to a jack or king. This is a type of exception-based description. The program cannot handle such exceptions. This is a problem for further research (see below).

4.2.4 Example 4. This is the only example which is not based on an actual game. The layout is taken from Abbott's rules for Eleusis [1]. The layout is shown in Figure 4 in section 3.2.1.1. The program discovered (in 381 milliseconds) two rules to explain the layout. Rule 1 is exactly the rule described by Abbott (except for the redundant suit selectors):

```
RULE 1: LOOKBACK: 1 NPHASES: 0 DECOMP
[MOD2(CARD1) = 1] => [SUIT(CARD0) = SPADES..CLUBS]
                     [COLOR(CARD0) = BLACK] V
[MOD2(CARD1) = 0] => [SUIT(CARD0) = DIAMONDS..HEARTS]
                     [COLOR(CARD0) = RED]

RULE 2: LOOKBACK: 1 NPHASES: 2 PERIODIC
PERIOD([VALUE(CARD0) = 2..8][VALUE(CARD0) <>VALUE(CARD1)]
       [VALUE(CARD0) = -VALUE(CARD1) + 4..8]
       [SUIT(CARD0) = CLUBS..HEARTS]
       [SUIT(CARD0) = SUIT(CARD1) + 0..2]
       [FACE(CARD0) = FALSE]
       [FACE(CARD0) = FACE(CARD1)]
       [PRIME(CARD0) <>PRIME(CARD1)][MOD2(CARD0) = 0]
       [MOD3(CARD0) = 1..2][MOD3(CARD0) = MOD3(CARD1) + 0..1]
       [MOD3(CARD0) = -MOD3(CARD1) + 0..1],

       [VALUE(CARD0) = 5..JACK][VALUE(CARD0) <>VALUE(CARD1)]
       [VALUE(CARD0) = -VALUE(CARD1) + 10..19]
       [SUIT(CARD0) = DIAMONDS..HEARTS]
       [SUIT(CARD0) = SUIT(CARD1) + 0..2][COLOR(CARD0) = RED]
       [MOD3(CARD0) = 1..2][MOD3(CARD0) = -MOD3(CARD1) + 2..0])
```

The second rule is worthless!

4.2.5 Example 5. The last example shows the upper limits of the program's abilities. During this game, only one of the human players even got close to deducing the rule, yet the program discovers a good approximation of the rule using only a portion of the layout that was available to the human players. Here is the layout:

```
4H  5D  8C  JS  2C  5S  AC  5S  10H
7C  6S  KC  AH      6C      AS
JH  7H  3H  KD
4C  2C      QS
10S 7S
8H  6D
AD  6H
2D  4C
```

The program was told, in this game, to check all three models. It produced the following rules after 6,538 milliseconds:

```
RULE 1: LOOKBACK: 1 NPHASES: 0 DNF
[VALUE(CARD0) < = 5][SUIT(CARD0) = SUIT(CARD1) + 1] V
[VALUE(CARD0) > = 5][SUIT(CARD0) = SUIT(CARD1) + 3]

RULE 2: LOOKBACK: 1 NPHASES: 1 PERIODIC
PERIOD([VALUE(CARD0) = VALUE(CARD1)-9]
       [VALUE(CARD0) = -VALUE(CARD1) + 4,5,7,11,13,17]
       [SUIT(CARD0) = SUIT(CARD1) + 1,2,3])

RULE 3: LOOKBACK: 1 NPHASES: 2 PERIODIC
PERIOD([VALUE(CARD0) = ACE,2,8,10]
       [VALUE(CARD0) = -VALUE(CARD1) + 1,8,9,10],

       [VALUE(CARD0) = 5..JACK][VALUE(CARD0) = VALUE(CARD1) + -0..6]
       [VALUE(CARD0) = -VALUE(CARD1) + 8..14]
       [SUIT(CARD0) = SPADES][SUIT(CARD0) = SUIT(CARD1) + 0..2]
       [COLOR(CARD0) = BLACK]
       [PRIME(CARD0) = PTRUE][PRIME(CARD0) = PRIME(CARD1)]
       [MOD2(CARD0) = 1][MOD2(CARD0) = MOD2(CARD1) + 0]
       [MOD2(CARD0) = -MOD2(CARD1) + 0][MOD3(CARD0) = 2]
       [MOD3(CARD0) = MOD3(CARD1) + 0][MOD3(CARD0) = -MOD3(CARD1) + 1])
```

The rule which the dealer had in mind was:

```
[SUIT(CARD0) = SUIT(CARD1) + 1][VALUE(CARD0)> = VALUE(CARD1)] V
[SUIT(CARD0) = SUIT(CARD1) + 3][VALUE(CARD0)< = VALUE(CARD1)]
```

It is very likely that a player could have deduced the correct rule once he/she had seen the rule produced by the program. The program has isolated the relevant variables, and has produced a very plausible description. Note that adding three to a suit gives the next lower suit in the cyclic interval domain of suits.

4.3 Evaluation

The Eleusis program, as it stands, is very capable of fitting data to the decomposition, DNF, and periodic description models. However, it is somewhat weak on rule evaluation, especially rule evaluation in light of knowledge of what makes Eleusis rules plausible. The program is surprisingly fast. During the design and implementation it was expected that memory would be the main constraint. Therefore, most routines were coded to tradeoff extra computation for less memory utilization. The fact that the program runs as fast as it does is very gratifying. The program does frequently exceed the available memory.

The following areas of the program could be improved:

a. Plausibility evaluation. As noted above, certain evaluation subroutines were not written. These should be written and installed.

b. Satisficing behavior. If the program is better able to assess the plausibility of the rules it is generating, it can cut off the search as soon as it has some plausible rules. This is the best form of effort control—far preferable to search limit parameters.

c. Rule filtering. Presently, the rules discovered by the program tend to contain redundant information. A knowledge-based filter should be developed which can remove these redundant selectors.

d. More general deductive mechanisms. Presently, the program conducts most of its deduction using a bit-string representation of the deck of cards. In particular, the PE of layer 3 uses knowledge about cards which is inappropriate. The lower layers should perform all deduction using VL_1 expressions (including intersection and complementation).

e. The current implementation has no ability to incrementally improve the rules it has discovered. An incremental learning capability should be included as part of the critic function.

4.4 Areas of Further Research

This thesis has covered only one small part of the problem of discovering plausible descriptions in sequential event sets. There are many problems remaining to be studied:

a. The problem of developing descriptions of noisy sequences needs to be investigated. There exist many real-world problems where noisy sequences are generated.

b. If an Eleusis "secret rule" involves some sort of exception (e.g. at the beginning), this program cannot discover it. In general, it is important that the program have the ability to handle rules which involve exceptions.

c. This thesis has concentrated on developing descriptions of a single sequence. There are interesting problems in which each event is a sequence of subevents and the task is to find patterns common to all of the events. This problem is very difficult, especially if each sequence of subevents is noisy.

d. One interesting problem faced by the designer of an Eleusis program is the problem of negative string plays. These cards are difficult to use during induction. In the present work, we are only able to perform an after-the-fact test for consistency. Work should be done to determine how these events can be used to assist the basic induction process.

e. The Eleusis program has not really been tested as a tool. A study should be undertaken to determine whether or not the program actually helps a person to play Eleusis more effectively.

f. The generality of the lower layers has not been tested either. Other applications involving noise-free sequential data should be identified. These would provide a test of the generality of the system.

5. CONCLUSION

A program has been developed which can serve as an intelligent assistant to a person playing the game Eleusis. The program has the capabilities to:

- discover rules which plausibly describe the layout
- accept rules typed by the user and test them against the layout
- extend the layout by suggesting cards to be played from the player's hand.

The program operates by transforming the input layout, through various interpretation steps, into unordered VL_1 events. The program then attempts to fit these events to one of three description models: decomposition, periodic, and DNF. Then, by various evaluation steps, the descriptions developed by model-fitting are checked and transformed for printout to the user.

The system is designed as a series of layers according to a knowledge discipline. As a result, the outer layers of the system may be removed and replaced by other knowledge-based layers which are specifically designed to perform in some other application area. This knowledge-layer architecture bridges the gap between the general-purpose learning algorithms used in the heart of the system and the special-purpose user-interface in the outer layer.

REFERENCES

[1] Abbott, Robert, "The New Eleusis," Available from Abbott at Box 1175, General Post Office, New York, NY 10001 ($1.00).

[2] Barto, A. G., J. M. Prager, "Forming Logically Simple Hypotheses in Parallel," Paper submitted to IJCAI-6, University of Massachusetts, Amherst, 1979.

[3] Box, G. E. P., G. M. Jenkins, *Time-Series Analysis: Forcasting and Control*, Revised Edition, Holden-Day, San Francisco, 1976.

[4] Buchanan, B. G., E. A. Feigenbaum, J. Lederberg, "A Heuristic Programming Study of Theory Formation in Science," in *Proceedings of the Second International Joint Conference on Artificial Intelligence*, 1971, pp. 40-48.

[5] Buchanan, B.G., D. H. Smith, W. C. White, R. J. Gritter, E. A. Feigenbaum, J. Lederberg, C. Djerassi, *Journal of the American Chemical Society*, 98 (1976) p. 6168.

[6] Buchanan, B. G., E. A. Feigenbaum, "Dendral and Meta-Dendral, Their Applications Dimension," *Artificial Intelligence*, 11 (1978) pp. 5-24.

[7] Buchanan, B. G., T. M. Mitchell, R. G. Smith, C. R. Johnson, Jr., "Models of Learning Systems," in *Encyclopedia of Computer Science and Technology*, J. Belzer, A. G. Holzman, and A. Kent, eds., Marcel Dekker, Inc., New York, 1977. (also available as HPP memo 77-39, Heuristic Programming Project, Stanford University, Stanford, CA).

[8] Chilausky, R., B. Jacobsen, and R.S. Michalski, "An applicaton of Variable Valued Logic to Inductive Learning of Plant Disease Diagnostic Rules," in *Proceedings of the Sixth Annual Symposium on Multiple Valued Logic*, Logan, Utah, 1976.

[9] Dietterich, Thomas G., R. S. Michalski, "Learning and Generalization of Characteristic Descriptions: Evaluation Criteria and Comparative Review of Selected Methods," *Proceedings of the Sixth International Joint Conference on Artificial Intelligence*, pp. 223-231, Tokyo, August 1979.

[10] Erman, L. D., V. R. Lesser, "A Multi-level Organization for Problem Solving Using Many, Diverse, Cooperating Sources of Knowledge," *Advance Papers of the Fourth International Joint Conference on Artificial Intelligence*, MIT, Cambridge, MA, 1975.

[11] Freuder, E., "A Computer System for Visual Recognition Using Active Knowledge," PhD thesis, AI-TR-345, The Artificial Intelligence Laboratory, MIT, Cambridge, Massachusetts, 1976.

[12] Gardner, Martin, "On Playing the New Eleusis, the game that simulates the search for truth," *Scientific American*, 237, October, 1977, pp 18-25.

[13] Hayes-Roth, F., "Collected Papers on the Learning and Recognition of Structured Patterns", Department of Computer Science, Carnegie-Mellon University, Jan. 1975.

[14] Hayes-Roth, F., "Patterns of Induction and Associated Knowledge Acquisition Algorithms," Department of Computer Science, Carnegie-Mellon University, May 1976.

[15] Hayes-Roth, F., J. McDermott, "Knowledge Acquisition from Structure Descriptions", In *Proceedings of the Fifth International Joint Conference on Artificial Intelligence*, 1977, pp. 356-362.

[16] Hayes-Roth, F., J. McDermott, "An Interference Matching Technique for Inducing

Abstractions", *Communications of the ACM*, 21:5, 1978, pp. 401-410.

[17] Hedrick, C. L., "A Computer Program to Learn Production Systems Using a Semantic Net," PhD. Thesis, Department of Computer Science, Carnegie Mellon University, Pittsburgh, Pa., 1974.

[18] Hunt, E.B., *Experiments in Induction*, Academic Press, 1966.

[19] Larson, J., "A Multi-Step Formation of Variable Valued Logic Hypotheses," in *Proceedings of the Sixth International Symposium on Multiple-Valued Logic*, Logan, Utah, 1976.

[20] Larson, J., and R. S. Michalski, "Inductive Inference of VL Decision Rules," *SIGART Newsletter*, June 1977, pp. 38-44.

[21] Larson, J., 'Inductive Inference in the Variable Valued Predicate Logic System VL21 : Methodology and Computer Implementation', Rept. No. 869, Dept. of Comp. Sci., Univ. of Ill., Urbana, May 1977.

[22] Lenat, D., "AM: An artificial intelligence approach to discovery in mathematics as heuristic search," Comp. Sci. Dept., Rept. STAN-CS-76-570, Stanford University, July 1976.

[23] Michalski, R. S., "Algorithm A^q for the Quasi-Minimal Solution of the Covering Problem," *Archiwum Automatyki i Telemechaniki*, No. 4, Polish Academy of Sciences, 1969 (in Polish).

[24] Michalski, R.S., "A Variable-Valued Logic System as Applied to Picture Description and Recognition," in *Proceedings of the IFIP Working Conference on Graphic Languages*, Vancouver, Canada, 1972.

[25] Michalski, R. S., "Conversion of Normal Forms of Switching Functions into Exclusive-Or-Polynomial Forms," *Archiwum Automatyki i Telemachaniki*, No. 3, Polish Academy of Sciences, 1971 (in Polish).

[26] Michalski, R. S., "DISCOVERING CLASSIFICATION RULES USING VARIABLE-VALUED LOGIC SYSTEM VL1," *Advance Papers of the Third International Joint Conference on Artificial Intelligence*, Stanford University, Stanford, CA, pp162-172.

[27] Michalski, R.S. "Pattern Recognition as Knowledge-Guided Induction," Rept. 927, Dept. of Comp. Sci., Univ. of Ill. Urbana, 1978.

[28] Michalski, R. S., J. Larson, "SELECTION OF MOST REPRESENTATIVE TRAINING EXAMPLES AND AN INCREMENTAL GENERATION OF VL1 HYPOTHESES: the underlying methodology and description of programs ESEL and AQ11," Report No. 867, Department of Computer Science, University of Illinois, Urbana, May 1978.

[29] Michalski, R. S., "Variable-valued logic and its application to pattern recognition and machine learning," In *Computer Science and Multiple-Valued Logic*, ed. D. C. Rine, North-Holland, 1977, pp. 506-534.

[30] Michalski, R. S., "VARIABLE-VALUED LOGIC: System VL1," *1974 International Symposium on Multiple-Valued Logic*, West Virginia University, Morgantown, West Virginia, May 29-31, 1974.

[31] Michie, D., "Measuring the Knowledge-Content of Programs," University of Illinois, Department of Computer Science Report UIUCDCS-R-76-786, May 1976.

[32] Michie, D., "New Face of AI," Experimental Programming Repts.: No. 33, MIRU, Univ. of Edinburgh, 1977.

[33] Schweizer, G. M., T. M. Mitchell, "Computer-assisted Structure Elucidation Using Automatically Acquired Carbon-13 NMR Rules," in ACS Symposium Series, No. 54, 'Computer-assisted Structure Elucidation,' D.H. Smith (ed), 1977.

[34] Soloway, E., E. M. Riseman, "Knowledge-Directed Learning," in "Proceedings of the Workshop on Pattern Directed Inference Systems," *SIGART Newsletter*, June 1977, pp 49-55.

[35] Soloway, E., "Learning = Interpretation + Generalization: a case study in knowledge-directed learning," PhD Thesis, COINS TR 78-13, University of Massachusetts, Amherst, MA., 1978.

[36] Vere, S.A., "Induction of Concepts in the Predicate Calculus," In *Advance Papers for the Fourth International Joint Conference on Artificial Intelligence*, 1975.

[37] Vere, S. A., "Induction of Relational Productions in the Presence of Background Information," In *Proceedings of the Fifth International Joint Conference on Artificial Intelligence*, MIT, Cambridge, MA., 1977.

[38] Vere, S. A., "Inductive Learning of Relational Productions", in *Pattern-Directed Inference Systems*, D.A. Waterman and F. Hayes-Roth (eds), Academic Press, 1978.

[39] Vere, S. A., "Multilevel Counterfactuals for Generalizations of Relational Concepts and Productions," Department of Information Engineering, University of Illinois, Chicago Circle, 1978.

[40] Waterman, D. A., "Serial Pattern Acquisition: A Production System Approach," working paper No. 286, Department of Psychology, Carnegie Mellon University, Pittsburgh, Penn., 1975.

[41] Winograd, T., *Understanding Natural Language*, Academic Press, 1972.

APPENDIX I

Input Grammar for Eleusis Program

This grammar describes the valid syntax of all commands and rules typed to the Eleusis program. The following differences should be noted between this grammar and that of VL_{21}. Firstly, this grammar permits functions and operators in the reference of a selector. The function may have an optional unary minus sign in front of it. Secondly, this grammar permits all selectors to use relations such as >=, <>, <, etc.

```
1    session ::= commandlist

2    commandlist ::= command
3        | commandlist ; command

4    command ::= HELP
5        | INDUCE
6        | EVAL
7        | PLAY
8        | Q
9        | CARD cardlist : ID
10       | UNCARD
11       | LIST listitem
12       | RULE vl2rule
13       | DELETE cardlist
14       | MINE cardlist
15       | STRATEGY ID
16       | KILL NUMBER
17       | DEFINE ID defdomain = deflist
18       | ADVICE advice
19       |          /* empty */

20   cardlist ::= SYCARD
21       | cardlist SYCARD

22   listitem ::= MINE
23       | STRATEGY
24       | ADVICE
25       | RULE
26       | ID              /* for other options */

27   vl2rule ::= segdefn ruledefn

28   segdefn ::= ID = simpleconjunct :  /* ID must be 'string' */
29       |    /* empty */

30   ruledefn ::= dcrule
31       | periodicrule

32   dcrule ::= conjunct
33       | dcrule V conjunct

34   periodicrule ::= PERIOD ( sconjunctlist )

35   sconjunctlist ::= simpleconjunct
36       | sconjunctlist , simpleconjunct
```

```
37  conjunct ::= simpleconjunct
38       | simpleconjunct => simpleconjunct

39  simpleconjunct ::= selector
40       | simpleconjunct selector

41  selector ::= [ referee rop reference ]

42  referee ::= ID ( varlist )

43  varlist ::= ID
44       | varlist , ID

45  rop ::= =
46       | <>
47       | >=
48       | <=
49       | >
50       | <

51  reference ::= value
52       | sign ID ( varlist ) moreref

53  value ::= valuelist
54       | NUMBER .. NUMBER

55  valuelist ::= valueentry
56       | valuelist , valueentry

57  sign ::= -
58       |    /* empty */

59  valueentry ::= NUMBER
60       | VALUE        /* a defined reference value */

61  op ::= +
62       | -
63       | +-

64  moreref ::= op value
65       |

66  defdomain ::= ( ID , NUMBER )
67       | ( ID )
68       |              /* empty */

69  deflist ::= def
70       | deflist , def

71  def ::= defvalue simpleconjunct

72  defvalue ::= ID
73       | NUMBER

74  advice ::= PARAMETERS params
75       | SEGMENTS sconjunctlist
76       | PLAUS plauslist
77       | DOMAIN domainlist
78       | ID tokenlist
```

```
79  tokenlist ::= token
80      | tokenlist token

81  token ::= ID
82      | NUMBER

83  params ::= param
84      | params , param

85  param ::= - parm1
86      | parm1

87  parm1 ::= NUMBER
88      | NUMBER ( NUMBER )        /* cost with tolerance */

89  plauslist ::= plaus
90      | plauslist , plaus

91  plaus ::= ID = NUMBER

92  domainlist ::= domain
93      | domainlist , domain

94  domain ::= ID = SYID
```

APPENDIX II

Eleusis Program Commands

Here is a synopsis of the user commands for the Eleusis program. The commands are broken down into five categories: layout management, managing the hand, managing the rule base, the learning element, and the performance element. The command input to the Eleusis program is free-format. Each command must be terminated by a semi-colon. When the program is ready for a command, it types:

```
ELEUSIS TOOL READY (nnn MS)
?
```

where nnn is the number of milliseconds required for the previous step. If a command is not yet terminated (e.g. missing its ';'), the program just prompts with a question mark. In the descriptions below, optional entries are placed in brackets.

Layout Management Commands

C[ARD] The CARD command adds a string of cards to the layout. One CARD command should be used for each turn of a player in the game. The syntax of the command is:

C[ARD] *cardlist* : *judgment*;

where *cardlist* is a list of cards of the form '2C' or 'QS' separated by spaces. *Judgment* indicates the dealer's judgment concerning the correctness of the play: 'Y' indicates the cards are correct, 'N' indicates the cards are incorrect.

U[NCARD] The UNCARD command is the reverse of the CARD command. It removes from the layout the string of cards added by the most recent CARD command. It may be used repeatedly to undo several CARD commands.

LIST LAYOUT This command lists the layout vertically along the page. The first card played is in the upper left-hand column. The correct cards (mainline) are in the left-hand column. The incorrect cards are listed across the page on the line of the correct card which they followed. Negative string plays (a string of cards declared to contain an error) are listed in parentheses.

Hand Management

M[INE] The MINE command (or HAND, a synonym) adds cards to the player's hand. The cards are simply listed after the command separated by spaces:

MINE AC 6S 9D JC 10C QS;

DEL[ETE] The DELETE command removes cards from the player's hand. The cards are simply listed after the command.

DELETE KC 2C;

LIST M[INE] Use LIST MINE to list the contents of your hand. See the section on the performance element below for an interpretation of the cards vs. rules matrix that is printed in the listing. The cards in the hand are listed along the left-hand column.

Rule management

R[ULE] The RULE command permits the user to enter a rule in VL_{22}. Refer to the grammar in Appendix I for details concerning rule syntax. An example of a RULE command is:

```
RULE PERIOD( [COLOR(CARD0) = RED],
             [COLOR(CARD1) = BLACK]);
```

Of course the rule can go on for more than one line. It is terminated by the ';'. When a rule is entered, it is immediately checked to see if it is consistent with the layout. This invokes the Critic function of the Eleusis program. A line will be printed indicating whether or not the rule is consistent with the layout. If the rule is inconsistent, some information concerning the source of the inconsistency is also printed. Then the rule is added to the VL_{22} rule base. The rule base is used by the performance element (see below).

I[NDUCE] The INDUCE command discovers rules that describe the layout and adds those rules to the rule base. Most of the relevant information is listed under the learning element below.

LIST R[ULES] Use LIST RULES to see the rules (and their assigned numbers) in the rule base. Rules remain in the rule base until deleted by the KILL command. Each rule is assigned a number. The number is used in the LIST HAND printout, and it is used to report information concerning the rule during the rule evaluation process.

K[ILL] The KILL command serves to delete rules from the VL_{22} rule base. Often a bad rule gets into the rule base, usually the result of an INDUCE command. To delete poor and implausible rules, a KILL command may be used. The KILL command accepts one number, the number of a rule to be deleted:

```
KILL 5;
```

This deletes rule number 5. No acknowledgment is printed. In order to determine the numbers corresponding to the rules, use the LIST RULES command.

The Performance Element

STRA[TEGY] The strategy used by the program is entered using the STRATEGY command. This strategy is used by the PLAY command to choose a card from the hand to play. There are two strategies. The CONSERVATIVE strategy directs PLAY to select the card which is legal under the largest number of rules in the rule base. The DISCRIMINANT strategy directs PLAY to select a card which will discriminate between the rules listed in the VL_{22} rule base. PLAY attempts to select a card which is covered by approximately half of the rules. It is wise to play DISCRIMINANT early in the game, and CONSERVATIVE after the first 30 cards have been played.

The strategy may be listed via LIST STRATEGY.

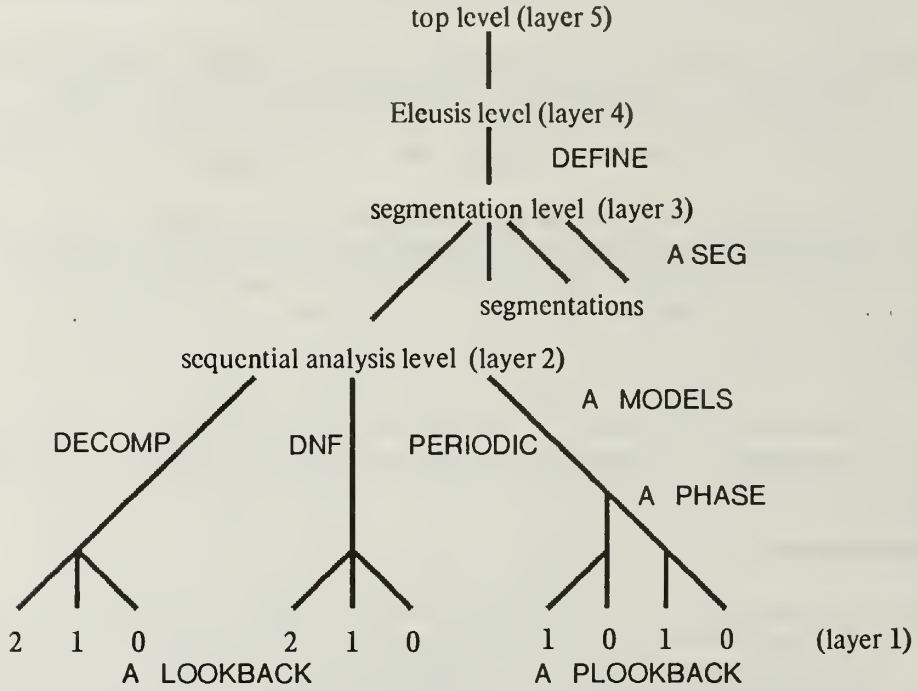
E[VALUATE] The EVALUATE command instructs the program to evaluate each rule in the rule base to determine what cards are currently playable under that rule. This information is then used to determine which cards currently in the player's hand are playable under each rule. This information can be printed out using the LIST HAND command. LIST HAND prints a matrix of cards in the hand versus rules in

the rule base. A 'Y' indicates that the card on that row is legal according to the rule in that column. The columns are numbered according to the rule numbers of the rules in the rulebase (See Chapter 4).

PLAY — The PLAY command instructs the program to choose a card to play according to the current strategy. See the STRATEGY description above for details of how the selection is made. The program makes the selection based on the most recent EVALUATE command. Thus, one should always precede a PLAY command by an EVALUATE.

The Learning Element

The learning element is the most complicated part of the program to use. The user must set up a collection of parameters which delimit the space of possible rules. When an INDUCE command is given, the program searches this space for plausible rules describing the main line. The search space looks like this:



The various notations attached to the tree are the relevant parameters which control the branching factor at that point in the tree. We examine the parameters from the perspective of the layers.

Layer 4 parameters

DEF[INE] — DEFINE may be used to add new descriptors to the program. The program initially only has knowledge of SUIT, VALUE, and LENGTH. To add color to the program, we would type:

```
DEFINE COLOR (NOMINAL, 50) =
    RED [SUIT(CARD0) = DIAMONDS, HEARTS],
    BLACK [SUIT(CARD0) = SPADES, CLUBS];
```

To add value modulo 2 to the program, we would type:

```
DEFINE  VALMOD2  (CLINEAR,  50)  =
    0    [VALUE(CARD0) = 2,4,6,8,10,12],
    1    [VALUE(CARD0) = 1,3,5,7,9,11,13];
```

The variables must be limited to 10 characters (only the first 10 characters are used). The notations NOMINAL and CLINEAR define the domain type of the variable. The "50" gives the plausibility for this variable (see below). After the = sign, we give a list of value-complex pairs separated by commas. Each value (either a symbol or a number) is defined by the VL_{22} complex which follows it. The complex may use any variables previously defined. The dummy variables used in the complex (in this case CARD0) determine what this new variable will be applied to (i.e. Cards or Strings).

LIST VARIABLES will list information concerning the variables which have been defined.

A[DVICE] DOMAIN This command can be used to change the domain of a variable. For instance, if we want SUIT to be nominal rather than clinear, we can write:

```
A DOMAIN SUIT = NOMINAL;
```

A[DVICE] GEN This command can be used to control the generation of derived variables such as DSUIT01 AND SVALUE01. A GEN gives a list of the types of derived variables which should be generated. The choices are SUM and DIFFERENCE. If we only want DIFFERENCE variables to be generated, we can type:

```
A GEN DIFFERENCE;
```

Each A GEN command completely replaces the list of information currently in the program.

Level 3 Parameters

A[DVICE] SEG[MENTATION] This command gives a list of all segmentation conditions the program should examine. As indicated in the diagram above, the null segmentation (left-most branch) is always investigated. Additional segmentations may be added by typing:

```
A SEG [COLOR(CARD0) = COLOR(CARD1)],
      [VALUE(CARD0) = VALUE(CARD1) + 1];
```

All segmentation conditions are selectors which have a variable in the reference. The segmentation condition must express the difference between two variables. A segmentation condition may have more than one selector in it. Segmentation conditions must apply to CARDs not STRINGs. The segmentation conditions given by each A SEG command completely replace the previous segmentation list.

A SEGPLAUS This controls pruning of unpromising segmentations. After each segmentation has been performed on the layout, it must satisfy two tests. First, the segmented layout must have at least *minsegplaus* number of events in it. Second, the segmented layout must have no more than *(maxsegplaus*size of unsegmented layout)*/100 events in it. The *minsegplaus* and *maxsegplaus* parameters are given as:

A SEGPLAUS *minsegplaus maxsegplaus*;

Level 2 Parameters

A MODELS This command tells level 2 and level 1 which models to investigate. The possible models are DNF, DECOMP, and PERIODIC. They are always investigated in that order. Example:

A MODELS PERIODIC DECOMP;

This tells the program to investigate only the decomposition and periodic models. Each A MODELS command completely replaces the previous setting of the MODELS list.

A LOOKBACK For the DECOMP and DNF models, the possible settings from the lookback parameter are determined by the A LOOKBACK command. The command provides two numbers, a minimum and a maximum lookback:

A LOOKBACK 0 2;

This gives the tree shown in the diagram above. To set lookback for periodic rules use:

A PLOOKBACK This gives the minimum and maximum lookbacks for periodic rules. Recall that a lookback in a periodic rule looks back to the prior occurrences of each phase, not on a card by card basis.

A PHASE This determines the possibilities for the number of phases to be examined for periodic rules. It is given as:

A PHASE *min max*

min must be at least 1.

Layer 1 parameters

The layer 1 parameters differ for each model. First we list the parameters applicable to the decomposition model, then to the DNF model. The periodic model has no additional parameters at this layer.

A COMPLEX This is a general parameter which applies to both the DNF and decomposition models. It indicates, for DNF, the maximum number of complexes that can appear in the solution. If the A^q algorithm has not found a solution before it reaches this quota, it gives up. For the decomposition algorithm, it indicates the maximum number of variables to be decomposed on (i.e. the maximum number of selectors to appear on the left-hand side of each if-then rule). It is specified as:

A COMPLEX 4;

A DEC This command specifies the parameters for the decomposition functional sort. See the body of the thesis for the meanings of the cost functions. They are entered in order of evaluation, with tolerances in parentheses:

A DEC 1(20),2,-3,4;

This specifies that cost function 1 is to be applied, with a tolerance of 20%. Then cost function 2 will be used. Then cost function 3 will be used, but first its value will be negated. Finally cost function 4 will be used to resolve ties still existing

after the first three cost functions have been applied.

A DECGEN This determines when the best trial decomposition is selected. If DECGEN is 0, the selection takes place immediately after the references are unioned. If DECGEN is 1, the references are generalized according to domain specific rules of generalization, and then the best decomposition is selected. If DECGEN is 2, overlapping selectors are removed, and then the best decomposition is selected. Example:

```
A DECGEN 1;
```

This is the recommended value for this parameter. If DECGEN is 2 and the rule does not fit the decomposition model, the program tends to run out of memory space.

For the DNF model, the following parameters may be used:

A AQ This sets the A^q cost functional. The cost functions and their meanings are:

1. Number of "new" events (events not covered by any previous star) covered by this complex in the set of positive examples.
2. Total number of positive examples covered by this complex.
3. Total number of negative examples covered by this complex.
4. Number of non-irrelevant selectors in this complex.
5. Sum of the costs of the non-irrelevant selectors in this complex. The cost of a selector is the plausibility of its variable subtracted from 100.
6. Number of non-irrelevant selectors that this complex has in common with the last complex on the MQ. This function is used to encourage the discovery of symmetric descriptions.

The cost functional is specified in the same way as the decomposition cost function above:

```
AQ 4(30),-1,3,-6;
```

A AQMAX This sets the MAXSTAR parameter for the A^q algorithm:

```
A AQMAX 6;
```

There is one other general set of parameters which control the adjustment phase of layer 1. These control the performance of the AQSTAR procedure when it is called during the adjustment phase:

A ADJ This enters the adjustment cost functional. The cost functions are the same as for the AQ cost functions above.

A ADJMAX This sets the MAXSTAR parameter for the adjustment process. It is entered in the same manner as A AQMAX.

The learning element is invoked by using the INDUCE command. New rules are discovered and added to the rule base. When the INDUCE command is completed, it executes a LIST RULES command automatically.

To list the various settings of these parameters, use the "LIST ADVICE" command. Also note that there are parallel settings for the LOOKBACK, PLOOKBACK, PHASE, and MODELS advice parameters for use with segmented rules (*viz* SEGLOOKBACK, SEGPLOOKBACK, SEGPHASE, and SEGMODELS).

For a list of legal commands, type H[ELP]. To exit the program, type Q.

BIBLIOGRAPHIC DATA SHEET	1. Report No. UIUCDCS-R-80-1024	2.	3. Recipient's Accession No.

4. Title and Subtitle

The Methodology of Knowledge Layers for Inducing Descriptions of Sequentially Ordered Events

5. Report Date: May 1980

6.

7. Author(s): Thomas Glen Dietterich

8. Performing Organization Rept. No.

9. Performing Organization Name and Address

Department of Computer Science
University of Illinois
Urbana, IL

10. Project/Task/Work Unit No.

11. Contract/Grant No. NSF MCS 79-06614

12. Sponsoring Organization Name and Address

National Science Foundation
Washington, DC

13. Type of Report & Period Covered

14.

15. Supplementary Notes

16. Abstracts This thesis describes an attempt to apply general induction techniques to the problem of discovering secret rules in the card game Eleusis. Eleusis is a card game in which players try to guess a secret rule (invented by the dealer) which describes a sequence of cards. A computer program was developed which has the capabilities to: discover plausible secret rules, accept rules typed by the user and test them against the cards played so far, and extend the sequence of cards by suggesting possible cards to be played from the player's hand.

Rule discovery is accomplished by fitting the data to three rule models. The raw data must be transformed by several knowledge-based processing layers before model-fitting can be performed.

A degree of program generality is obtained by the use of a knowledge layer programming methodology, in which the functions of the program are segregated into layers according to the generality of the knowledge they required. This allows the program to be applied to similar tasks merely by "peeling off" and replacing its outer layers.

The thesis demonstrates that general inductive techniques can be used to solve complex learning problems, but they form only part of the solution. In the Eleusis domain, data interpretation, rule evaluation, and model-directed induction were all required in order to develop a satisfactory program.

17. Key Words

machine learning
computer induction
model-fitting
variable-valued logic
computer assistants
knowledge acquisition
knowledge-based systems
programming methodology

17c. COSATI Field/Group

18. Availability Statement	19. Security Class (This Report) UNCLASSIFIED	21. No. of Pages 57
	20. Security Class (This Page UNCLASSIFIED	22. Price

JUN 3 1981

BOUND BY

UNIVERSITY OF ILLINOIS-URBANA
510.84 IL6R no. C002 no.1021-1027(19
Report /
3 0112 088397010